FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Shellhive: Towards a Collaborative Visual Programming Language for UNIX Workflows

Omar Alejandro Castillo de Castro

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira, Assistant Professor

Second Supervisor: Tiago Boldt Sousa, Assistant Lecturer

August 5, 2014

Shellhive: Towards a Collaborative Visual Programming Language for UNIX Workflows

Omar Alejandro Castillo de Castro

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor João Manuel Paiva Cardoso
External Examiner: Doctor Ricardo Jorge Silvério Magalhães Machado
Supervisor: Doctor Hugo José Sereno Lopes Ferreira
August 5, 2014

Abstract

UNIX-based operative systems, provide tools that allows users to create workflows that can be used to process data. However, not only they are difficult to learn, it is very difficult to maintain long workflows using such tools. In this thesis, we propose a solution to leverage such UNIX tools, to empower users with little experience in programming with the ability to create workflows that are easier to understand and modify. The application itself, allows the user to design workflows collaboratively in order for people who are comfortable in UNIX environment to help each other and potentially learn from the more experient users. We also create various concepts that allows the reusability of multiple workflows, increasing the maintainability of the workflows. The purpose of the application is to leverage a standardized command-line language provided by UNIX operative systems, the "UNIX Shell", using a visual programming language. It uses visual representations of workflows which are interacted by users, that are then automatically translated to a UNIX Shell compatible code.

We start with the background related to the work, explaining some of the paradigms, including basic knowledge about some of the concepts of Unix Shell. We then analyze commercial solutions already deployed as well as a scientific approach of creating visual dataflows, in order to understand their strengths and weaknesses, so that they could be used in our solution. Next, we describe the problems that the application ought to solve and the contributions that this thesis expects to create. Next, we describe the solutions created, explaining the logic and reasoning behind each of the features implemented in the application. We also created tests to validade our hypothesis. Testing a visual programming language is similar as testing an user interface, it's difficult to test, except by collecting direct feedback from end-users. Using the results from a set of quasi-experiments, it will be possible to proceed with incremental improvements in the final solution. Finally we close the content of this dissertation with a conclusion of this document, the contributions that were made, including the publication of an article to the CDVE conference, and as well the future of the created application.

Resumo

Os sistemas operacionais baseados em UNIX fornecem ferramentas que permitem os utilizadores criarem fluxos de trabalho que podem ser usados para processar dados. No entanto, não só são difíceis de aprender,como é muito difícil manter workflows longos usando as ferramentas. Nesta tese, propomos uma solução para aproveitar tais ferramentas UNIX, para capacitar os usuários com pouca experiência em programação com a capacidade de criar fluxos de trabalho que são mais fáceis de entender e modificar. A própria aplicação, permite os utilizadores criam fluxos de trabalho de forma colaborativa, para que as pessoas que se sintam confortáveis no ambiente UNIX ajudem uns aos outros e, potencialmente, aprendam com utilizadores mais experientes. Também criamos vários conceitos que permitem a reutilização de múltiplos workflows, facilitando a manutenção dos fluxos de trabalho. O objetivo da aplicação é aproveitar uma linguagem *standard* da linha de comandos fornecido por sistemas operativos UNIX, o "UNIX Shell", usando linguagens de programação visual, usando representações visuais de fluxos de trabalho para ser interagido por utilizadores, os quais são automaticamente traduzidos para código compatível com UNIX Shell.

Começamos com o *background* relacionado com o trabalho, explicando alguns dos paradigmas, incluindo conhecimentos básicos sobre alguns dos conceitos do *Unix Shell*. Depois analisamos soluções comerciais existentes, bem como uma abordagem científica da criação de fluxos de dados visuais, a fim de entender seus pontos fortes e fracos, de modo que poderiam ser usados na solução. Em seguida, descrevem-se os problemas que a aplicação pretende resolver e as contribuições que esta tese espera criar. A seguir, descrevemos as soluções criadas, explicando a lógica e raciocínio por trás de cada uma das funcionalidades implementadas na aplicação. Foram criados testes para validar as hipóteses da dissertação. Testar uma linguagem de programação visual é semelhante a testar uma interface com o utilizador, é difícil de testar, a não ser através da recolha de feedback direto dos utilizadores finais. Utilizando os resultados de um conjunto de quase-experiências, será possível prosseguir com melhorias incrementais no solução final. Finalmente, fechamos o conteúdo desta dissertação com uma conclusão deste documento, as contribuições que foram feitas, incluindo a publicação de um artigo à conferência CDVE, e assim o futuro do projeto criado.

Acknowledgements

I would like to express my gratitude to my family, who supported me through the whole course, without them, I would not be doing a dissertation in the first place.

I have to thank Professor Hugo Sereno Ferreira for accepting the dissertation and for getting some free time when I really needed, and Tiago Boldt Sousa, that could always have some time to give me fast feedback and accurate answers to my needs. I would like to thank my friend Luis Fonseca for sharing ideas about the dissertation, some of his ideas were bizzare, but others allowed for the enrichment of this dissertation.

At last I express my gratitude to my deceased father who not only supported my family, but also, he was the one that helped me the most through my academic life.

Omar Castro

To my family.

Contents

List of Figures

LIST OF FIGURES

List of Tables

re

Abbreviations

API	Application Programming Interface
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OS	Operating system
SSH	Secure Shell
UML	Unified Modeling Language
VCS	Version control system
VP	Visual Programming
VPL	Visual Programming Language

Chapter 1

Introduction

This chapter introduces the technical report of this dissertation. It starts by describing the context related of this dissertation following the motivation behind it. Then, it identifies the problems addressed in this thesis, and describes its objectives and goals. Lastly, an explanation about the report structure is written.

1.1 Context

The amount of data is growing exponentially. As time passes, this growth hampers its management using traditional tools [datb]. The difficulty can be related to various factors: data capture, storage, search, sharing, analytics and visualization, etc. [SS12]. According to IBM [IBM14], when managing the data life-cycle of big data, organizations should consider 3 factors: volume, velocity and complexity of big data.

- **Volume**. As of 2012, approximately 2.5×10^6 GB of data are generated daily, which can impact the total cost of ownership for data warehouses and other big data environments if data growth is not managed appropriately. It is estimated that data will grow exponentially, as shown in figure 1.1, that in 2020 it will be generated nearly 45×10^9 GB of data [data].
- **Velocity**. Big data environments support time-sensitive processes, such as analyzing 500 million daily call detail records in real-time to predict customer churn faster [IBM14].
- **Complexity**. Organizations capture a variety of data formats and comb through massive data sources in real-time to analyze and identify patterns within the data. For example, to identify fraud for credit card customers, to identify financial trends for investment organizations, predict power consumption for energy companies, etc..

These factors motivated the creation of new programming paradigms and architectures to solve problems related to big data. One of said paradigms is *stream programming*, which was initially designed for media and image processing applications. Then it has successfully grown into a more general-purpose programming model. [GCTR08, EAG+07]

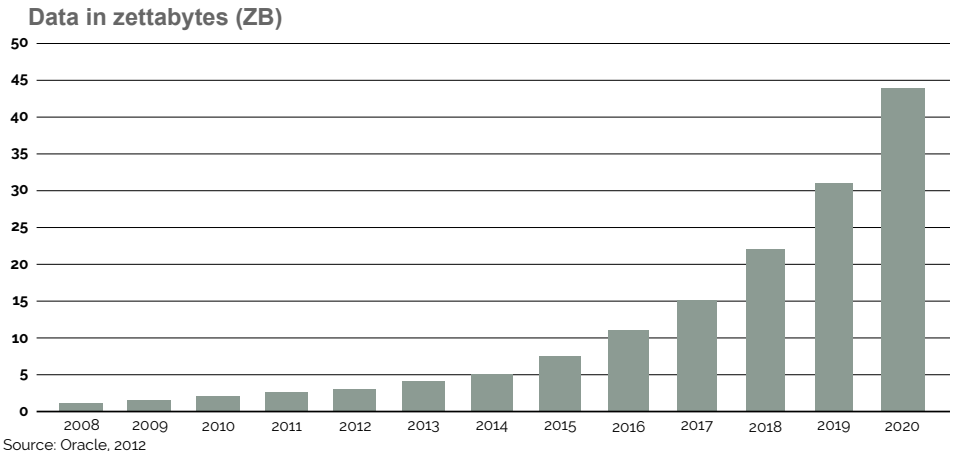

Figure 1.1: Estimation of the amount of data generated yearly through the years.

In fact, these models were studied for more that 30 years. There even exists programming languages that are based on some of those models such as the Unix command language [Bou78], which is included in the Unix based operative systems (OS) for a long time.

1.2 Problem

Unix Shell is a programming language and a command language [Bou78]. It is the core concept of some Unix command interpreters, such as *csh*, *bash*, *ksh* and others. Those interpreters have a common short-coming, it is very difficult to design complex big-data workflows due to two main reasons:

- **Steep learning curve**. The user not only has to study about the functionality of each command, but has to learn the idiosyncrasies of some of its commands, namely, the `sed` and `awk`. There exists other problems that hampers the user to learn about the utility of those commands, like inconsistencies in the naming of the arguments of some commands, some sections of the command manuals ("man pages") can be confusing, the input or output is often untyped or unchecked, etc..

- **Hard to maintain workflows**. While it may not apply in small workflows, it becomes evident that is difficult to maintain large workflows due to the increasing difficulty of creating readable code. When we include a decent amount of commands, specially when some commands requires the usage of special parameters such as adding scripts as arguments, they may need to include escape characters for the workflow to function correctly.

1.3 Motivation

There are reasons to believe that the defined problems can be solved using a programming paradigm called *visual programming* (VP). It is known that conventional programming languages are difficult to learn and use [LO87]. There were studies to create concepts that reduces barriers to learn about programming. One of the concepts was to use visual expressions to create and maintain scripts [Mye90a]. This concept originated the VP paradigm.

Also, the Unix environment contains a great set of utilities called “UNIX tools". Most of those utilities are included on Unix-based operating systems. Unix dates back to 1969 [Sys], because of that, most of the tools had many years of improvement, creating new tools and improving the old ones. Due to those improvements, the Unix tools have little to no dependencies since most of the commands are included in Unix-based OS. They are also lightweight and have a respectable performance because many of the commands were created in a time where computers were not as powerful as they are today. Also, most of the commands are developed using programming languages that focus on execution efficiency such as C and Assembly.

As time passes, new tools were created that allows the execution of workflows in more conventional ways:

- **Commands can be executed remotely**. With the inclusion of the Secure Shell (SSH) [YL06], we can execute commands in a remote computer.
- **Commands can be executed in parallel**. The `parallel` command allows an efficient execution of multiple tasks in parallel by distributing the workload between various processors inside a machine. A basic example is the processing of large amounts of files, a computer with multiple processors can utilize each processor to process a subset of files, instead of using one processor to process all the files.
- **Commands can be executed in multiple computers**. The same *parallel* tool can distribute the workload with multiple machines, using the SSH tool for remote execution.

1.4 Main Goals

The main goals of this dissertation are to explore the capabilities of those tools to process large quantities of data, and to explore the advantages of applying VP to create and maintain Unix command-line scripts.

These goals are to be achieved by creating a web platform that will be used as an alternative to the Unix terminal for big-data related purposes, which:

- **Allows the synthesization of text code though a visual model.** The application should be able to generate code through a model and vice-versa , this method is designed as round-trip engineering [Aß03, HLR08]. This way it allows the execution of Unix commands trough the designed model.

- **Have a better maintainability than writing commands with text.** With the application, it should be as easy or easier to create, modify and interpret workflows compared to the conventional usage of textual Unix commands, so that it can be used by less experienced programmers.
- **Can be designed by multiple users collaboratively.** In other words, the application should allow multiple user to design in the application, remotely or not, which every change to the model is notified by other users using the same model.

The described application is directed to a public that is interested in processing large amounts of textual data, and wants to leverage standard Unix tools for that purpose. It is also directed to users that are being used to process data using Unix tools (e.g. system administrators, that uses UNIX Shell in a daily basis).

It aims to allow users to design data processing tasks in a way that should be easy to understand even to the public that has little to no experience with programming. It was implemented by following an iterative development methodology, with emphasis on good software engineering practices such as coding standards, continuous testing and refactoring.

One innovative aspect is that we are leveraging several characteristics, namely “zero-dependencies” and “lightweightness". It also becomes a powerful educational tool since the end-result should be the synthesization of what the user would write in the command prompt.

1.5 Report Structure

The remaining content of this dissertation is organized into three parts, with the following overall structure:

1. **Background & State of the Art**. The first part reviews the most important concepts and issues relevant to the thesis.

   The next chapter 2 (p. 7) provides a literature review on software engineering paradigms, techniques and tools related to the dissertation.

   Chapter 3 (p. 17) provides an overview of the current state-of-the-art, and ideas that could be used in the development of the application.

2. **Problem & Solution**. The second part states the problems researched and the proposed solutions.

   The 4th chapter (p. 25) describes the problem to be addressed by this project.

   Chapter 5 (p. 29) contains an overview of the the solution implemented, including its architecture, as well as the reasoning behind the design and technological decisions taken.

3. **Validation and Conclusions** . The third part describes the experiments made for the validation of the solution, and presents the conclusions of the dissertation:

Chapter 6 (p. 47) provides information about the quasi-experiments to test our application.

The last chapter 7 (p. 53) provides the conclusion of this technical report with the work plan of the project.

Chapter 2

Background

This chapter describes the research relevant to the context to this dissertation. We start by describing the area in concepts, starting with a background information about the current area and then giving a basic information about the relevant concepts and tools of the Unix Shell.

2.1 Software engineering

Laplante [Lap07] defines software engineering as:

> *"A systematic approach to the analysis, design, assessment, implementation, test, maintenance and re-engineering of software, that is, the application of engineering to software. In the software engineering approach, several models for the software life cycle are defined, and many methodologies for the definition and assessment of the different phases of a life-cycle model"*

It encompasses all aspects of conceiving, communicating, specifying, designing and maintaining software systems. It includes activities related with production of artifacts related to software engineering such as documentation and tools [Lap07], which allows the creation of reliable and efficient software applications.

The importance of this area is the impact it gives on large, expensive software systems and the role of software in safety-critical applications, by integrating significant mathematics, computer science and practises whose origins are in engineering. [sof].

There are two areas of software engineering that are relevant to the context of this dissertation: *Visual Programming* and *Stream Processing*.

2.2 Visual Programming

Myers defined Visual programming (VP) as "any system that allows the user to specify a program in a two (or more) dimensional fashion", stating that "textual language is one dimensional

because the compiler or interpreter processes it as a long, one-dimensional stream" [Mye86]. Visual Programming Language (VPL) is a programming language that includes visual expressions. Visual expressions can be described as conventional flow-charts and graphical programming languages [Mye86]. In short,VPL tries to combine the communication power of a Visual Language with the possibilities of a programming language.

The author believes that the idea of creating a paradigm such as VP isn't new, because images were used as a mean of communication many times throughout history. Even today, in software engineering, we still use it. An example of a visual description of a problem instead of a textual one is the Unified Modelling Language (UML). UML provides a set of visual tools to represent solutions, and it is used world-wide by expert programmers. One of the earliest examples of visual programming may be the GRaIL system [NoF14, hol] in 1968.

2.2.1 Concepts

In 1994 Burnett suggested a classification for this kind of languages [BB94]. However, this classification was too specific. Boshernitsan et al. launched in 2004, made a summary of the visual programming types, classifying them in: Purely Visual Languages, Hybrid Text and Visual Systems, Programming-by-example systems, Constrained- oriented systems and Form-based system [BD04].

Characteristics of a Visual Language

VPLs have a set of common approaches [Bur01]. Those approaches, or strategies, create a set of characteristics that are important in this kind of languages:

- **Concreteness**. Some languages are powerful because of abstraction. However, concreteness goes in the opposite direction. Visual programming languages tend to be concrete, using real values, contrasting with types of values.
- **Directness**. Instead of working with possible values or objects the user works directly with a concrete value or object.
- **Explicitness**. In textual languages, the relation between objects is implicit in VPLs. The more explicit those relations are, the better. A user of a VPL should have the most explicit information possible.
- **Immediate Visual Feedback**. Changes in the visual programming language should be seen and felt immediately, without any type of compilation.

2.2.2 Classification scheme

Although Boshernitsan's summary [BD97] is useful to characterise several systems, the name given at programming-by-example can be misleading. According to Brad A. and Myers, example-based programming has two forms: *programming by example* and *programming with example* [Mye90b].

Therefore, the term example-based systems may be more general than the proposed term by Boshernitsan.

- **Purely Visual Languages.** This VPL is compiled directly from the visual form. Most, if not all, interactions are done in a visual environment.

- **Hybrid Text and Visual Systems.** The visual form of this system is a layer that is translated to a text form. In this kind of systems, it is possible to program in a text or a visual form and alternate between visualization modes.

- **Example-based systems.** Example-based-systems can be divided in: Programming by example and programming with example. Those systems are based in examples of input and output. Programming by example systems, or "automatic programming", "tries to guess or infer the program from examples of input or output or sample traces of execution" [Mye90b]. Programming with example systems, "require the programmer to specify everything about the program (there is no inference involved), but the programmer can work out the program on a specification example. The system executes the programmer's commands normally but remembers them for later reuse" [Mye90b]. This kind of systems can be seen as a macro-building systems.

- **Constrain oriented systems** In this kind of systems, there is a set of constrains that are built to create rules at a certain environment. This technique is useful to construct simulations, dynamic documents, and manipulable geometric objects. [BD97]

- **Form-based systems**. The most known example of this system is spreadsheets. This system is characterized by the presence of cells that have some sort of connection between them. [BD97]

Example-based systems are out of the dissertation scope because the purpose of this dissertation is to leverage a command language, not create a machine learning language. Constrained-oriented systems may be useful to construct simulations, dynamic documents, and manipulable geometric objects. It doesn't focus on data flows, which is one of the concepts of the architecture of the Unix shell. Form-based systems may provide a metaphor of cells and connections, but lack context of data flows.

Hybrid system is the system that it's going to be developed in this dissertation since one of its goals is to synthesize the Unix commands.

Purely Visual Languages are a great approach since we deal with the visual layer only, however, the purpose of this thesis is to have a layer on top of the Unix code. One type of these systems describes the best way to represent a stream programming language, which will be used as the visual layer: *visual dataflow programming*.

2.2.3 Visual Dataflow Programming

A dataflow language is an applicative language that bases upon the notion of data flowing from one function entity to another or any language that directly supports such flows [DK82], the graph structure of the dataflow allows a simpler evaluation of the program. The function composition should also be easier to understand [AA92].

Dataflow programming appeared as a method to do parallel computing. It appeared as a theoretical concept and then researches tried to implement it in hardware. From those projects, several diagrams appeared to illustrate the solutions developed and those diagrams were used to represent information on visual dataflow programming languages. However, data-flow programming faced problems representing conditional execution and iteration. That representation was possible but added comprehension issues essentially when the complexity of solutions started to increase (e.g. representing loops with conditions to finish it, also representing the same with inner loops (a loop inside a loop)). Therefore, dataflow programming languages sacrificed parallelism for comprehensibility, creating a class of those languages called "Controlled Dataflow Languages" [CG11].

Visual Programming does allow to interpret the information on how we intend to create a dataflow visually, however, this area doesn't focus on how the data will be flowing in the model, so we will introduce another area of software engineering that is related to this dissertation: *stream processing*

2.3 Stream processing

A stream processing application, organizes software into basic units of first-in-first-out (FIFO) data streams flowing through stream filters. [BL13]

A data stream is a list of data records. Each record in a stream is a collection of related data words. A stream filter, or a computation kernel, is a small program which consumes data from a single input data stream, and produces data on a single output data stream.

This model can be analogized as a factory, where raw materials are transformed into a final product by passing through machines. The raw material is considered as a record before being processed in the first filter the final product being the data record coming from the output stream of the last filter, and each machine is considered a stream filter that processes records of data.

By exploiting data parallelism in the records of a stream using tream processing, it allows the process of large amounts of data, which is commonly required in media applications and large databases.

2.3.1 Concepts

The applications that make use of a stream abstraction are diverse, with targets ranging from embedded devices, to consumer desktops, to high-performance servers [TKA02], these kind of applications can be referred to as *streaming application*

Characteristics of a stream based application

There exists a set of common properties that allows the characterization of streaming applications [TKA02]. These properties are:

- **Large streams of data.** Stream programs can operate on a large sequence of data items. Data streams generally enter the program from some external source, and each data item is processed for a limited time before being discarded. A normal application has a large degree of data reused from the input set. Normally, the processing of the input stream on a filter starts as soon as the first records of stream data arrive and as soon as possible, evicts the first records of the output stream.
- **Independent stream filters.** Each filter reads one or more items from an input, performs some computation and writes one or more items to an output stream, it doesn't care about the state of another filter, it only cares about the data that he receives to read.
- **A stable computation pattern.** A certain set of filters are arranged in a regular, predictable order to produce an output stream that is a given function of the input stream. The arrangement generally remains stable during the execution of the streaming application.
- **Occasional modification of stream structure.** Even though each arrangement of filters is executed for a long time, there are still dynamic modifications to the stream graph that occur on occasion. One occasion can be when a wireless network interface experiences a high noise on an input channel, it might react by adding some filters to clean up the signal.

This model relates with a concept that has been a part of UNIX and other operating system for quite some time, which is called "Pipes and Filters" [SLR10].

2.3.2 Pipes and Filters

The Pipes and Filters (PaF) architecture simply contains *pipes* and *filters*. [SLR10]

A filter, is a component that processes data from its input, and produces data from its output like explained above. A pipe is a connection that moves the data emitted by one filter to another for consumption.

The Unix-based OS contains a command line interface that can be used to create workflows that bases on this architecture. [Bou78].

A filter is a component with the same concept. It reads from an input interface, computes it, then writes to the input interface.

Pipes are typically provided as separate components such as data repositories or queues, whose sole purpose is to provide interfaces that allow putting data into and getting it out. A pipe is only responsible for transmitting data between filters; it does not carry out any processing of data.

The main characteristic of the PaF architecture is total isolation of each component. Each component, let it be pipe or filter, works independent of any other component. Since filters are isolated, they do not know their predecessors or successors.

2.3.3 Stream programming

Stream processing advocates a gather, operate, and scatter style of programming. [GR04] First, the data is gathered into a stream from sequential, strided, or random memory locations. The data is then operated upon by one or more computation kernels, where each kernel comprises of several operations. Finally, the live data is scattered back to memory.

This style of programming makes explicit the parallelism and is locality present in applications. This parallelism is encoded within computation kernels. If kernels are data-parallel, they can be broken down into mutually independent chunks which can be executed in parallel. [GCTR08]

Although many applications can potentially be expressed using the stream programming style, some of the desirable program characteristics include: large amounts of data to operate on, high arithmetic intensity, memory accesses that can be determined well in advance of their use and producer-consumer locality between computation kernels. Stream programming, initially targeted only for media applications, has evolved into a more general programming paradigm. [RDK+98] There is another programming paradigm related to this one: Functional Reactive Programming.

2.3.4 Functional Reactive Programming

Functional reactive programming (FRP) is a paradigm for programming hybrid systems, that is, systems containing a combination of both continuous and discrete components, in a high-level, declarative way [HCNP03]. One of the main abstractions of RFP is the notion of time flow [NCP02].

Concepts

The key ideas in FRP are its notions of two concepts: *signals* and *events*

- **Signals**, previously called *behaviors* [NCP02] are values that varie over time.
- **Events**. An event can be understood as an action that occurs in a single, discrete point in time, having little or no duration [NCP02]; one example of an event is a mouse click. Events can be understood as time-ordered sequences of discrete-time event occurrences. [WH00].

We can relate this with stream programming, where a signal is a data record that goes from the output of a component to an input of the next component. An entry of a data record to a stream filter is an event, which the filter reacts to, and immediately processes the data record, creating another signal to the next component, which is a data record sent from its output.

The next topic is about Unix Shell, a command language and a programming language that uses some of the principles of stream programming. The topics contain basic knowledge about some of the concepts and tools, which are going to be referred in the next chapters.

2.4 Unix Shell

The Unix shell is both a command language and a programming language that provides an interface to the UNIX operating system [Bou78].

As a programming language it contains control-flow primitives and text-valued variables. As a command language it provides a user interface to the process-related facilities of the Unix OS.

As explained in section 2.3.3 the Unix command language can advocate the stream programming model, a Unix concept called pipes.

2.4.1 Concepts

Here we will explain about four concepts of the Unix commands that will be used in the dissertation: *redirection, pipes, command substitution and process substitution.*

Redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing it. The example code 2.1 saves the output of the command to a file.

```
1 ls -l > result.txt #saves result of ls -l into file result.txt
2 ls -l >> log.txt   #appends result of ls -l into file log.txt
3 wc < data.txt      #counts characters, words, and lines of file data.txt
```

Listing 2.1: Three redirection examples, with their current explanation in the comment.

The notation > file is interpreted by the shell and is not passed as an argument to ls. If the file does not exist, the shell creates it; otherwise, the contents of the file are replaced with the output from the command. To append to a file, the notation >> is provided. Similarly, the standard input may be taken from a file by writing for example the third command on the listing 2.1.

Pipes

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by |, as in the code 2.2 which shows two ways to connect 2 processes.

```
1 ls -l | wc
2 ls -l > file; wc < file
```

Listing 2.2: Two examples of counting the characters, words, and lines of the result of ls

The difference is that that no file is used in the first command. Instead, the two processes are connected by a pipe that is created by an operating system call. Pipes are unidirectional;

synchronization is achieved by halting wc when there is nothing to read and halting ls when the pipe is full. This matter is dealt with by the Unix OS instead of the shell.

Command substitution

The standard output from a command can be substituted in a similar way to parameters. The command pwd prints on its standard output the name of the current directory. For example, if the current directory is `/usr/home` then the two commands in 2.3 are equivalent because the output of the command `pwd` is the current directory (`/usr/home`).

```
1  d= $(pwd)
2  d=`pwd`
3  d=/usr/home
```

Listing 2.3: Equivalent Unix commands if the current directory is "/usr/home"

The entire text between grave accents `` `...` `` is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions, except that the grave accents must be escaped using a backlash `\` . Another way is to use the dollar sign which is commonly used for variable substitution, advanced interpreters added a feature on the interpretation that the dollar sign followed with text between parentheses `$(...)` would be interpreted as a command line. This would remove the need of escape the right parenthesis in the text in case its followed by a left parenthesis.

Process substitution

Process substitution is an unique feature not included in the basic Unix shell [bas]. It allows to redirect the output of a process to the input of multiple processes, and redirect the output of multiple processes to the input of a single process.

To exemplify, let's say that you had two versions of a program that produced large quantities of data. You want to see the differences between the output from each version. You could run the two programs, redirecting their output to files, and then comparing the files with the diff utility.

Another way would be to use process substitution. There are two forms of this substitution. One is for input to a process: `>( command )`; the other is for output from a process: `<( command )`.

`command` is a process that has its input or output connected to a named pipe. A named pipe is a temporary file that acts like a pipe with a name. In our case, we could connect the outputs of the two programs to the input of the application via named pipes:

```
1  diff <(prog1) <(prog2)
```

Listing 2.4: Prints the difference between the output of two commands

`prog1` and `prog2` are run concurrently and connect their outputs to named pipes. The diff utility reads from each of the pipes and compares the information, printing any differences as it does so.

2.4.2 Tools for parallel and remote execution

As shown in section 1.3, there exists tools that allow the parallel and remote execution of the application, even to distribute the workload with multiple machines.

The next code shows an example of running a command-line code remotely using the `ssh` command:

```
1 ssh user@remote-host "gzcat eventlist-21-30.txt.gz | sort -uk1,2 --buffer-size=7G |
     gzip > sorted-21-30.txt.gz"
```

Listing 2.5: sort the contents of a compressed file in a host computer

We can see that the command line code is inside the quotation marks. The remote computer is identified by a hostname, in this code the hostname is “user@remote-host”

The next code shows an example of running a command-line code in parallel using the `parallel` command:

```
1 find eventLists2013/ -name 'eventlist.*.gz' | parallel "gzcat {} | sort -uk1,2 --
    buffer-size=7G | gzip > {.}.sorted.txt.gz"
```

Listing 2.6: Sort the contents of compressed files inside the direcotry “eventLists2013” filtered by file name ’eventlist.*.gz’ in parallel

Again, the command line is inside the quotation marks. The parallel command will execute the command for each line written in the `find` command, replacing “” inside the code with the line received from the input of the parallel command, and replacing “.” with the same line but without the text that comes after the dot. In this case it is replaced by the name of the file without the extension part.

The next code shows an example of running a command-line code, to distribute the workload with multiple machines using the `parallel` command

```
1 find eventLists2013/ -name 'eventlist.*.gz' | parallel --sshlogin server.example.
    com,server2.example.com --sshlogin server3.example.com --sshlogin : --trc {.}.
    sorted.gz "gzcat {} | sort -uk1,2 --buffer-size=7G | gzip > {.}.sorted.gz"
```

Listing 2.7: Prints the difference between the output of two commands

The explanation is the same as before, however, the parallel command contains sshlogin parameters which defines a host computer to distribute the workload, if we use the parameter, the parallel command distributes the workload in remote computers. To include the client in the list

of machines to run the application, we add a colon in the sshlogin parameter list. The usage –trc parameter means that the files will first be transferred to the remote computers, then it retrieves the results of the commands which resulted in the creation of a file. After retrieving the file, it cleans the remote computer, removing the files generated during the execution of the command.

2.5 Conclusions

This chapter focused on theoretical concepts of the related work of this dissertation and a basic knowledge about concepts of Unix shell. Visual Dataflow programming is a great approach because it has a great potential at providing a simple and intuitive syntax. It also has a relatively straight-forward representation.

We are leveraging the usage of Unix commands with paradigm and concepts defined in this chapter, such as the stream programming paradigm

Chapter 3

State of the Art

This chapter describes the existing state-of-the-art relevant to the context of the problem. While many solutions exists, it was decided to find a considerable number of solutions. While they are relevant to the context of the problem, we can take ideas to implement into our solution.

3.1 NoFlo

NoFlo is a Flow-Based Programming environment for JavaScript [nofb]. In flow-based programs, the logic of your software is defined as a graph. The nodes of the graph are instances of NoFlo components, and the edges define the connections between them. The figure 3.1 shows an example of an application along with its result at the bottom right. [nofa]

Concept

There are two main concepts defined in NoFlo:

- **Components:** a component is a block that defines a function that can take inputs and produces an output, the components contains another output for error handling.
- **Packets:** packets or messages are blocks of data that come from an output of a component, to an input of a connected element.

Functionalities description

Some nodes require some sort of giving input for cases there are no edges to connect to an input port just like in figure 3.2, if there were no text in the "selector" text field it would show an error for not having mandatory arguments.

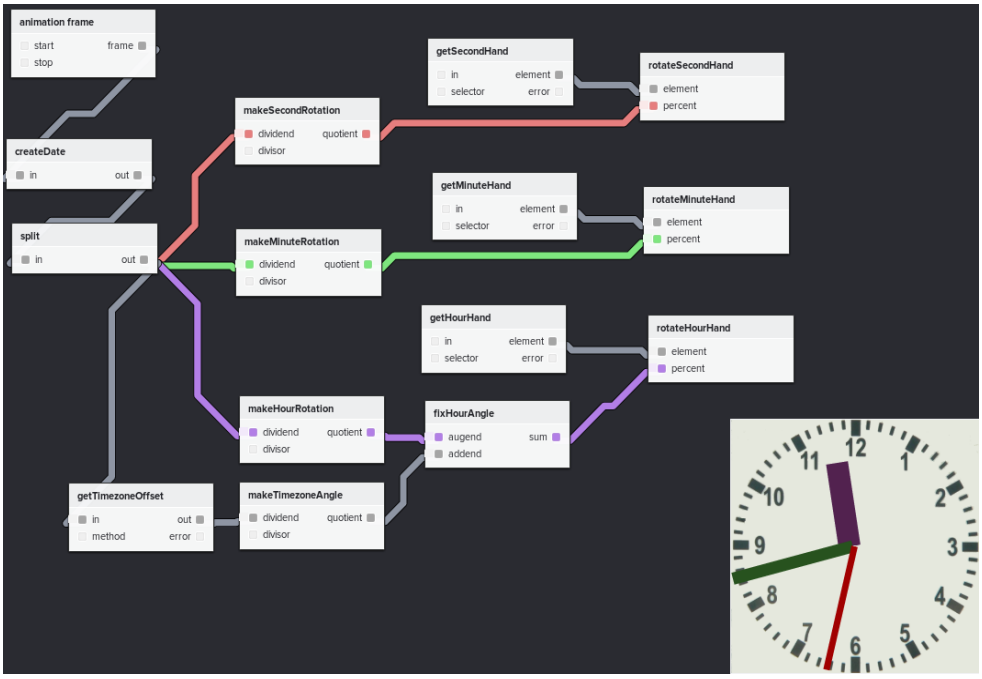

Figure 3.1: Analog clock implemented using NoFlo.

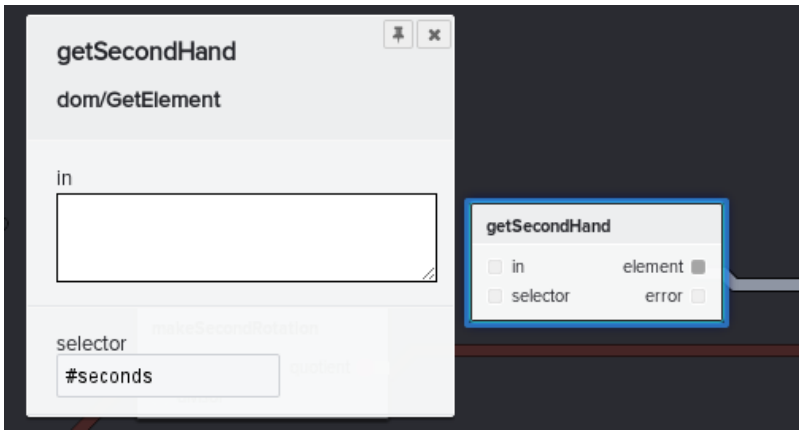

Figure 3.2: A GUI node interaction in NoFlo

Lessons learned

Simplifying the input/output types seems a good approach. It is important that the end-user can see the types just by looking on a node. Combinations between different types can be useful but can also be dangerous. Without information, an end-user can fall in a coordination barrier or understanding barrier. Another important thing is the ability to group nodes, this feature can be a good approach to solve the scale-up problem associated with visual programming languages.

3.2 Blender Composite Nodes

Blender Composite Nodes, while not being a full application but a feature of Blender application, empowers end-users with a tool that allows them to build some sort of a visual script that can be applied in their scenes. The application wiki [Ble14] gives an insight into the features provided by Composite Nodes.

> *Compositing Nodes allow you to assemble and enhance an image (or movie) at the same time. Using composition nodes, you can glue two pieces of footage together and colorize the whole sequence all at once. You can enhance the colors of a single image or an entire movie clip in a static manner or in a dynamic way that changes over time (as the clip progresses). In this way, you use composition nodes to both assemble video clips together, and enhance them.*

An example of the representation used can be seen in figure 3.3. The viewer node is a node used to print the output on the screen in a panel, that is not shown on the figure.

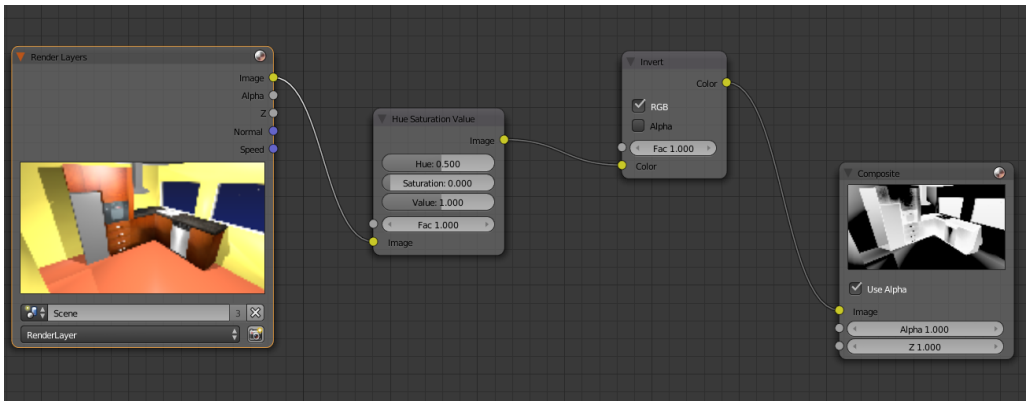

Figure 3.3: An image being transformed into inverted grayscale image using Blender Composite Nodes

Concepts

There are three main concepts defined in Composite Nodes: Nodes, Noodles and Node Groups.

- **Nodes**. A node is a block that can be seen as a function that can take inputs and can produce a set of outputs. There are three types of nodes: Input Nodes, Processing Nodes and Output Nodes [Ble14]. Input Nodes exist to produce information. An integer value is an example of a Input Node. Processing Nodes can apply filters or transformations on their inputs to produce a set of outputs. Filters like blur or contrast are examples of a processing node. Output Nodes are useful to finalize a composition and specify were the result will be saved. A viewer node to display the output or a file output node are examples of those kinds of nodes.
- **Noodles**. A node has configurable parameters, and its outputs can be connected with the inputs of other nodes creating a network called Noodle.
- **Node Groups**. Nodes can be grouped into a single node creating a Node Group that can be connected with other nodes.

Functionalities description

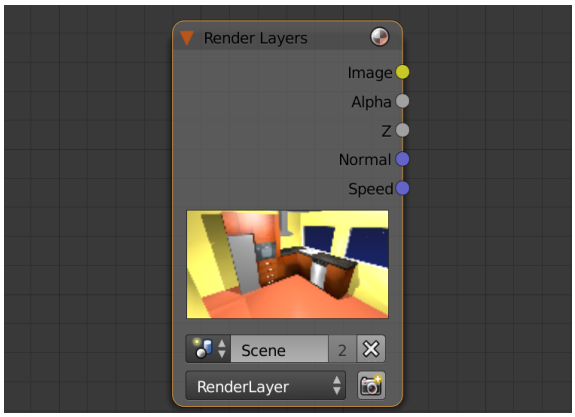

Figure 3.4: Possible colors on the sockets of a node

Output sockets should connect with input nodes of the same type, therefore, to identify sockets types Blender Node Composition defined a color for each type (fig 3.4). There are three types available: RGBA - yellow; three-dimensional vector - blue; value - grey. When different socket types are connected a default conversion will occur as follows:

- **Vector to Value** - Average (X+Y+Z)/3.0;
- **Color to Value** - Grayscale conversion (0.35*R + 0.45*G + 0.2*B);
- **Value to Vector or Color** - Copies value to each channel;

- **Color to Vector** - Copies RGB to XYZ;
- **Vector to Color** - Copies XYZ to RGB and sets A to 1.0.

Lessons learned

The usage of a graphical interface inside a node for each different type of node is a good approach, it is also important that the user has the possibility to hide the interface so that it doesn't block the view to other nodes. In any case it becomes difficult to manage the interface with the nodes, the interface can be used as a separate component outside the model, just like figure 3.2 in the previous state-of-the-art application.

3.3 IShell

IShell is a visual command line interface where files, commands and applications as well as user-specified commands are represented by icons. The abstract of the research paper created by Kjell Borg [Bor90] gives a better definition of the application:

> *IShell is a visual user interface for interaction using gestures under the UNIX operating system. It uses a visual script language for building commands called IScript and it is an integral part of the IShell environment The user can directly describe and execute pipelined command sequences using gestures. The user is constantly guided by visual cues. [Bor90]*

Concept

There are three main concepts defined in IShell, machines, detectors and edges.

- **IShell machines** An IShell machine in a graph is a single node that each "machine" is differentiated by its icon.
- **Detectors** or views are instruments that can be connected to an edge. They allows the user monitor the flow of information in the edge. The content of a stream is not affected by the detectors although the pace of the execution might change.
- **Edges.** They bind the filters together into pipelines.

Lessons learned

While the usage of icons is a good approach to identify the icons at first sight, it becomes difficult to understand the functionality of the node due to the lack of text. The monitoring of the data inside a stream is a great approach to see the result of each node in case the user makes an error in the design, it becomes easier to find and correct mistakes and errors in the design of the task,

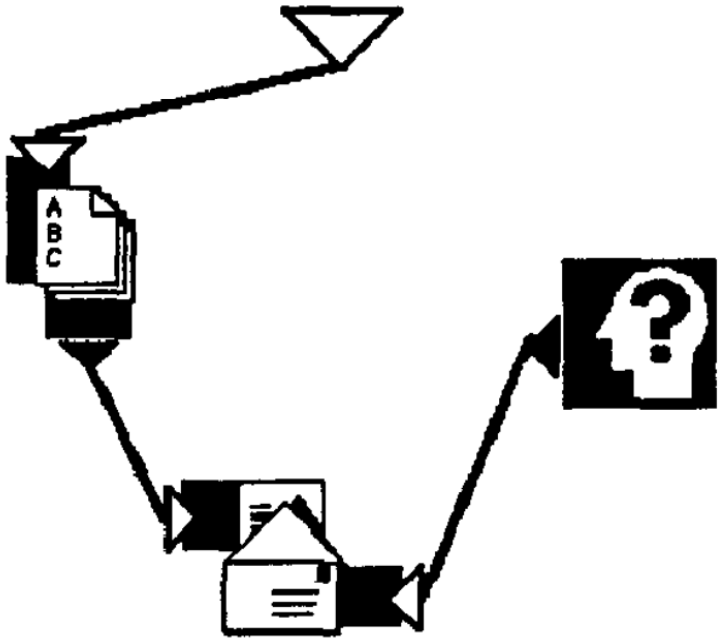

Figure 3.5: A IScript source code graph example.

however, monitoring large is tedious, but it still is a great approach when testing a workflow with small amount of data before testing with large amounts of data.

3.4 Yahoo Pipes

The Yahoo documentation [pipb] introduces the application as

> *A free online service that lets you remix popular feed types and create data mashups using a visual editor. You can use Pipes to run your own web projects, or publish and share your own web services without ever having to write a line of code".*

Concept

There are two main concepts defined in Pipes, modules, pipes and sub pipes.

- **Module** A module can be considered a node in a dataflow graph, a block that can be seen as a function that can take inputs and can produce an output, the modules are grouped by their functionality: *Sources* are data sources (such as Yahoo Search) that returns a RSS feed, *User Inputs* are input fields that your Pipe's users can fill at run-time, *Operators* have basic features to modify the values of an output of a module pipe, *Url*, *String*, *Date* contains modules for building and manipulating URLs text and date respectively. [pipa]

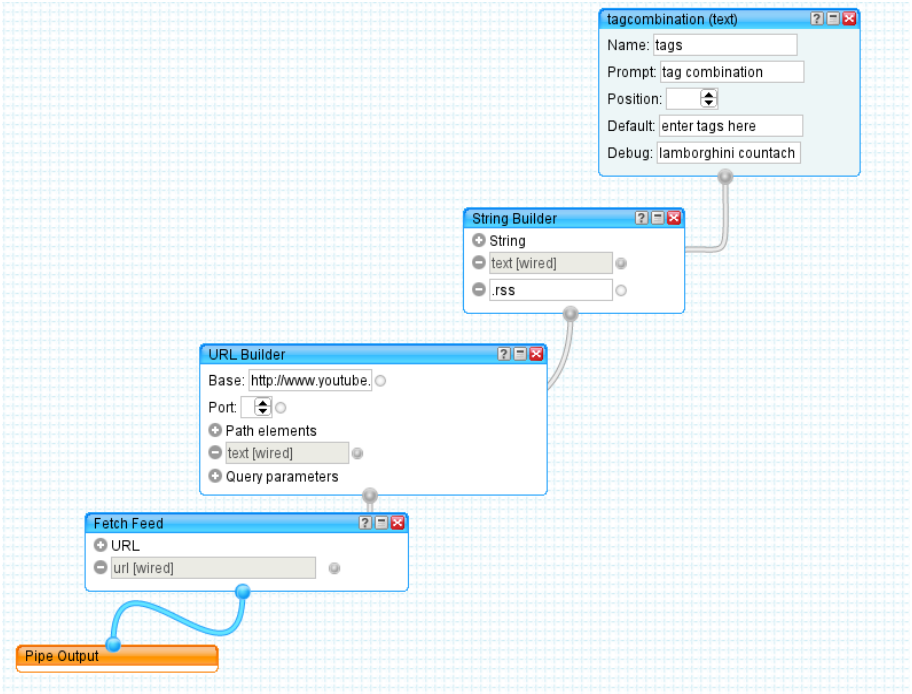

Figure 3.6: Getting a data from youtube of tags in Yahoo! Pipes

- **Pipe** can be viewed as a graph, it contains modules, sub-pipes, and the connectors
- **SubPipe** A subPipe behaves just like a regular module, with the addition of a button link. Clicking it opens a new tab in the editor where you can edit the subPipe on the fly. It can be viewed as an encapsulated Pipe in a module.
- **Terminal** terminal can be viewed as a port which connects two modules.

Functionalities description

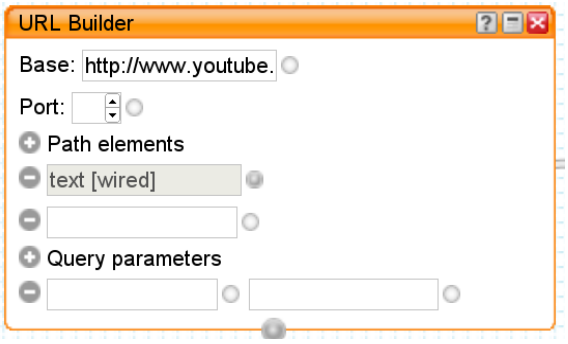

Figure 3.7: Terminals inside a module, the terminals are the circles in front of the parameters, the white colored ones isn't connected, while the gray are.

Input terminals can only be connected with output terminals that uses the same type of data. Each module have attributes that can have a terminal (fig 3.7). It can be connected with an output terminal of another module.

Lessons learned

Some of the Unix commands have a considerable amount of parameters, using a way to connect a output to a parameter is a good approach to use the command substitution feature of the Unix shell.

3.5 Conclusions

In this chapter we discussed the state of the existent concepts. NoFlo is the closest concept to what we want to explore. It allows collaboration between multiple users, integrates well with the JavaScript platform. We also intend to implement an interface for specific commands like the way Blender Composite Nodes and Yahoo! Pipes implements it, that way we can solve problems related with the idiosyncrasies of different Unix commands. The way IShell uses to identify the nodes in a relatively fast way may be good approach, however, use of different user interface makes this approach rather redundant, also the way NoFlo uses colors to identify an edge path is another great approach when the model becomes so big it becomes difficult to manage it.

Chapter 4

Problem Definition

As described in the main goals (section 1.4), the goal of this dissertation is to explore the capabilities of Unix tools to process data. It is done by leveraging the the Unix Shell using VP.

As seen in the previous chapter, none of the current state of the art focuses on data processing in a visual form. The proposed solution should focus in the simple creation and modification of tasks to process large quantities of data. The solution can be improved if it allows multiple users to use the application collaboratively, allowing multiple user to design a workflow, remotely or not, where every change made in the model is notified immediately by other users currently working with the same model.

4.1 Thesis statement

It is the author belief, that most of the problems users and developers face when creating tasks with the Unix Shell, could be bypassed by using a visual layer that helps users to interpret the purpose of the workflow.

As such the author claims the following hypothesis:

> *When end-users are provided with a real-time visual collaborative workflow editor to process large amounts of data using Unix command-line tools, they will significantly increase their efficiency to design and modify workflows. Such application would make a powerful educational tool since it allows the synthesization of what the end-user would write in the Unix terminal.*

This statement uses terms whose meaning may not be consensual, and therefore leads to questions that deserve further discussion:

1. What should be understood by *real-time visual collaborative workflow editor*?

We can divide this expression in 3 parts: *collaborative workflow editor*, *Real-time collaborative workflow editor*, and *visual workflow editor*.

Collaborative workflow editor, is a tool that allows multiple users to create and edit workflows collaboratively, where multiple users can edit the same workflow at the same time.

The **real-time** part means that changes in the model a user is working will be noticed by connected user working on the same model. There exists other types of collaborative tools, such as a version control system (VCS). In a VCS, changes made by a user will be sent to a server when he explicitly sends them. As long as he does not send the changes, connected users cannot see them.

By **visual workflow editor**, we mean that workflows are not created nor edited using text, but using visual representations of a workflow, such as a graph.

2. What should be understood by *Unix command-line tools*?

Unix operative systems contains a set of tools that allows the user to interact with the machine. The most common tools are the console, a UNIX Shell interface that execute commands, a command language, and a set of small programs that will be executed by the language.

3. How is *efficiency* measured?

Efficiency is to be measured by the quantity of work required to create and modify models that executes a desired task, achieving a desired effect. For measuring efficiency, it will be considered both the velocity, that is, the amount of work per person per unit of time, and the amount of changes needed to achieve the necessary effect.

4. How would the application make an *educational tool*?

Allowing the translation of visual expressions to Unix Shell code and vice-versa can reduce barriers of end-users to learn about the language. Being a collaborative application, inexperienced users can learn from other experienced users about the language when working with the same workflow, as well as learning the advantages of the usage of stream processing concepts to process data. Making the prototype suitable for educational purposes.

5. Who are the *end-users*?

In the context of a visual programming tool, both computer scientists from beginners to experts and users with little background on the usage of Unix commands are to be considered end-users. Therefore, end-users are defined as the group of persons who will ultimately use the application.

4.2 Expected Contributions

The primary outcome of this thesis encompasses the following aimed contributions to the body of knowledge in software engineering:

- **The formalization of a visual language for systems whose domain is the usage of command languages**. The main goal of this thesis is to explore the capabilities of a command language by leveraging it using a VP paradigm.
- **Exploration of a way to use the standard Unix mechanism**. Provides a collaborative UNIX Shell programming platform, which generates and executes command.
- **The collaborative aspect of editing and sharing such designs with other peers**. The solution allows mutilple users to create and modify scripts collaboratively, allowing each peer, to know the changes made by other peers immediately.

Chapter 5

Solution prototype

As seen in the background chapter (ch. 2), we were dealing with Visual programming and Stream programming paradigms. In this case, the use of a Visual Dataflow Programming model was proposed as it is one of the most fitting models to represent a stream based application.

The proposed solution was a web based framework for creating dataflow-based processing using common UNIX Shell commands, abstracted as connected block components. A user creates components with a set of inputs that transforms incoming data, making the result available in the outputs. These outputs can be connected with inputs of other components, creating a model that is similar to a data-flow diagram. This approach was implemented as a collaborative web application as a drag-and-drop editor to create and connect components. A parsing engine then converts the diagram into a UNIX Shell script that can be executed in a UNIX-based OS. The project is open source[1], codenamed *ShellHive*.

5.1 Overview

A set of concepts were defined to transmit ideas for the end-users.

Component

A component is an abstraction of a block that is commonly used in the software engineering. One example of that usage is black box testing. The idea behind black box testing is: We want to test a functionality, but we don't want to understand what the program is doing. We only know that giving a specific set of inputs should create a correspondent collection of outputs.

Ports

A port can be considered as a connector inside a component where the end of a connection is located. There are 2 types of ports: input and output. A component will transform the information received from their input ports, and send the results to their output ports. An output port can

[1]The code is available at https://github.com/OmarCastro/ShellHive

connect to several inputs, also each input can be connected to several outputs. The ports are located on different sides of the components. The inputs are located on left side, and the outputs on the right side.

Connection

A connection represents a *pipe* in the context of the UNIX Shell language. Visually, they represent a line that connects two components. There are times that using pipes does not suffice, like joining the result of 2 commands. In this case, the connection would represent a redirection to a *named pipe*, where it would be used as an input to a component.

The connections are mostly typed, in other words, a connection should be created if it passes the following requirements:

- **The ends of the connection does not belong to the same component**. A component should not connect to itself. Not only they are difficult to parse, they can be dangerous, as they can create infinite cycles.
- **The connection should connect different type of ports**. An output port can only connect to an input port and vice-versa. A command can only read the output of another command, or the contents of a file, it should not read the input of another command, if the objective is to send the output of a component to multiple components, just connect the output of a command to their respective components.
- **The connection does not create a cycle in the workflow graph**. Having cycles in the graph can create infinite cycles. The commands would run continuously without stopping, wasting available resources.

Macros

The larger and complex is a workflow, the less maintainable it becomes. One of the ideas to increase the maintainability of the workflow was to group multiple components and connections to a single component, thus the macro concept was created to test in the prototype. A macro is a composition of interconnected components. It can be represented as a sub-graph with various points of entry and exit. It allows the user to reuse groups of tasks, improving the maintainability of the workflow.

The entry points and exit points are located in 2 special components: one component represents the inputs and the other the outputs of the macro. The number of entry and exit points are variable and can be customized for the user needs.

5.2 Architecture

The prototype architecture contains various interconnected modules that identifiy the application. The figure 5.1 presents a layered architecture of the application. Most of the logic of the application is on the server. The client mostly provides an interface to interact with the workflow, communicating with the server with the help of Web Sockets technologies and frameworks. Being a web application, the global architecture can be divided in two architectures: the server and the client.

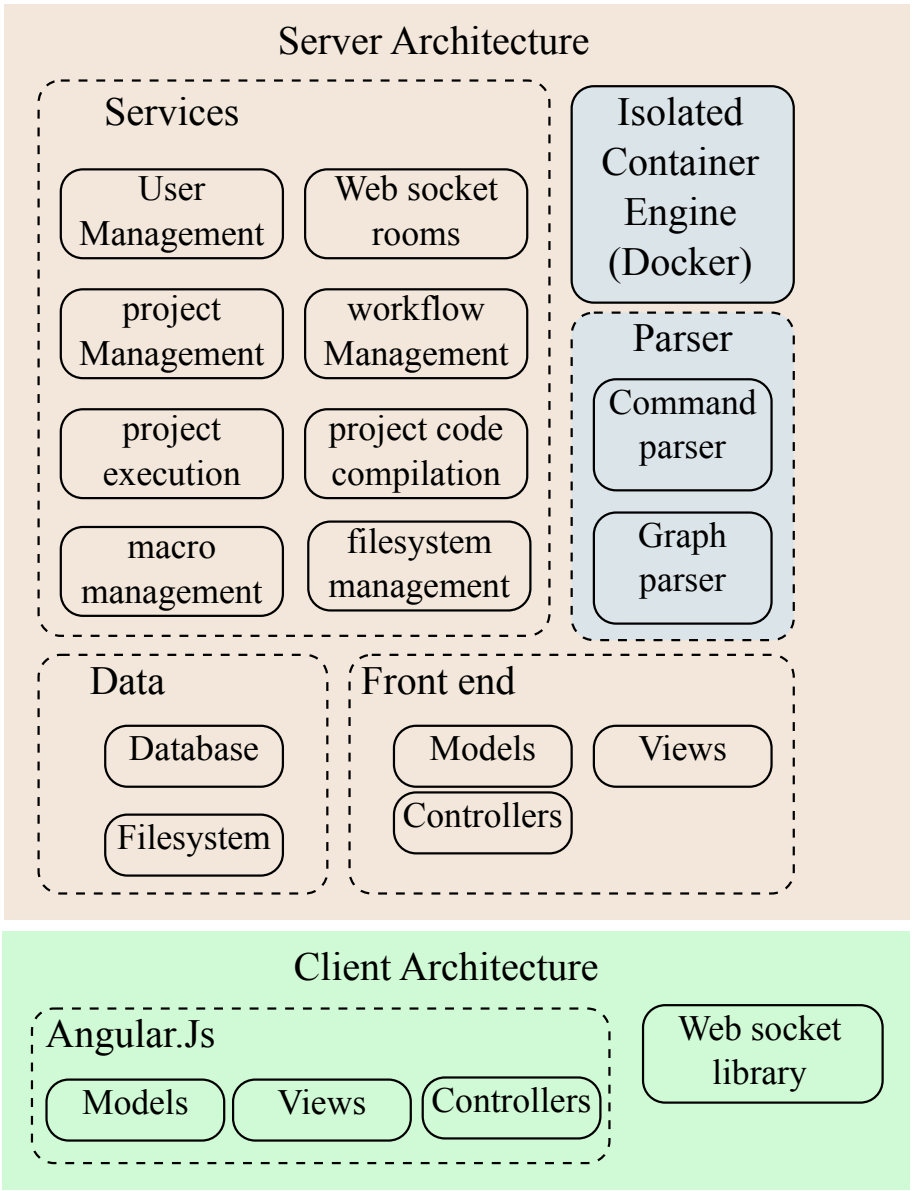

Figure 5.1: Layered architecture of the application.

5.2.1 Server Architecture

The server was developed with Sails.js, a Node.js framework, adopting the Model–view–controller (MVC) pattern, it contains multiple sub-modules that come from various external libraries integrated in each other, allowing for a faster development of the application. The server architecture can be divided in 5 modules:

Service. This is the main component, it manages the prototype business. This component connects to the other 4 components. It manages the users in a project, manages the creation, modification and deletion of projects, as well as its files and workflows.

Data. The Data layer is responsible to manage the data in the server. The sails framework includes API's to connect to various database management systems. The prototype data is saved in a MySQL database. The database contains information about users, workflow, macros and projects, however, it does not contain project files. The files are saved in an internal filesystem in the application. This allows the isolation of the filesystem when executing a workflow.

Front end. The application was made using an MVC pattern, the front end is mostly responsible for the communication with clients.

Parser. The parser module provides a framework responsible for the translation of the UNIX Shell code to its graphical representation. This framework was developed separated from the other modules, so it can be used as a standalone application. This module was developed using a parser generator called Jison [Jis].

Container Engine. The container engine is responsible for the isolation of the execution of the commands. Before executing a command, the engine initializes a container with the project folder. The folder content is updated after it's execution.

5.2.2 Client Architecture

The client module provides the front-end application that is interacted by end-users, using a web browser. The module was developed using frameworks and libraries that use web standards. That way, the prototype is supported in the most known web browsers.

The client side was developed using *Angular.js*, a JavaScript framework maintained by Google that assists with running single-page applications [Anga], and a set of libraries and utilities that depend of it, such as AngularUI [Angb].

The client communicates with the server using Socket.io, a web socket framework that focuses on bidirectional event-based communication.

5.3 Implementation Details

This section shows details of the implemented features.

5.3.1 Script compiler & generator

Some of the end-users are used to writing scripts and commands in a command-line interface. Generating graphs using a command line improves the speed of designing workflows because the user will have two ways to generate graphs: creating components and connecting them manually, or generating a graph automatically with a command line. The latter one can benefit end-users that are used to write commands with a keyboard.

The script compiler translates the Shell code into a data format that can be represented visually in the web browser interface. The script generator does the inverse, it parses visual information and compiles it to a UNIX Shell script that can be executed in the server. The generated code contains single quotes in the arguments of the commands when needed.

Since the UNIX shell does not interpret text inside single quotes, it ensures that unsafe scripts are not running when injecting code in command arguments when using command substitution techniques, improving the security of the application.

5.3.2 Graph Layout

One of the features that improves the usability of the application is the automatic creation of components and connections by using a single command line. However, when creating a graph from a command line, specially those with multiple commands, one of the problems we had to tackle is how to position the nodes. A quick study to find an adequate layout style to position the components was done.

There were various layout styles found [Lay14]. After studying them, we decided to implement an algorithm to position the nodes based on the hierarchical layout style [Hie]. One of the primary purposes of the hierarchical layout style is to highlight a flow within a directed graph. It is the layout that fitted us the most because the workflow is visually represented as a directed graph, highlighting the flow of data from a component to another during its execution.

5.3.3 Cycle detection

Having cycles in a graph creates a loop with no end, called “infinite cycles”. When this happens, the commands will continuously run without reaching an end, and the execution process would need to be forced to stop to free the resources used in the execution. By removing cycles, it removes the possibility of existing infinite cycles, thus ensuring that the execution of the generated scripts reaches to an end.

Cycle detection was implemented to ensure that no cycles appears in the workflow. The algorithm to detect cycles is done by sorting topologically the nodes of a directed graph. The algorithm normally stops and fails when a cycle exists.

When searching for a library to check for cycles in a graph, we found only one and it used this algorithm. When sorting the nodes of a graph, the library throws an error pointing that the graph contains cycles. All we had to do is to check for that error, saving development time for other important tasks.

5.3.4 List of commands

There was the need to study each command because many commands treat their inputs and options in a unique way. For that reason, it was necessary to limit the number of commands to be available in the prototype. Also the priority of adding new commands was always low.

The choice of the available commands was based on the simplicity to represent them visually. Some commands, like *perl*, are not available because they are general programming languages, and they are difficult to abstract visually.

Another metric used to choose the commands was based on parse simplicity. To know how to parse a code of general programming, it is required to study the grammar of each language. Studying a programming language takes a lot of time, so it was not feasible due to the limited time required to develop the prototype.

The table 5.1 shows the developed list of available commands[2] in the prototype. There are less commands than planned. However, the number of available commands was sufficiently large to be tested by users.

5.3.5 Execution isolation

Every command is executed in an isolated environment. They will only modify the content of the project directory they are being executed to, leaving anything outside the directory unmodified. Isolating the environment increases the security of the application, because a user will have access to a virtual operative system which looks like as if he would have access to a real OS. After a workflow is executed, everything except the project directory will be reverted to the initial state of the execution. Meaning created files will be removed, deleted files will be recovered, etc..

The isolation environment is done using Docker, a container engine. In contrary to a Virtual Machine (VM), it does not initialize an entire OS just to execute a workflow. Instead, the Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in a user-space on the OS, making the initialization of a container very fast, without losing noticeable performance.

The figure 5.2 shows a sequence diagram of the execution of a workflow. When the user requests an action to compile the workflow, the request does not contain the command to be executed. Instead, it contains an identification of the workflow to compile. The server then queries its data based on the identification to translate it into a Shell script. When the translation is successful, the server gets the location of the project directory and sends to the Docker, along with the compiled script, to create an isolated container and execute such script in the project directory.

[2]With the description taken from their current manuals

Table 5.1: Developed list of allowed commands.

Command	Description
awk	pattern scanning and processing language.
bunzip2	decompress bzip2 files.
bzcat	decompresses bzip2 files to stdout.
bzip2	a block-sorting file compressor.
cat	concatenate files and print on the standard output.
curl	a tool to transfer data from or to a server. The command is designed to work without user interaction.
date	print or set the system date and time.
diff	compare files line by line.
grep	searches the named input files (or standard input if no files are named, or if a single hyphen-minus (-) is given as file name) for lines containing a match to the given pattern.
gunzip	decompress gzip files.
gzip	reduces the size of the named files using Lempel-Ziv coding (LZ77).
head	output the first part of files.
ls	list directory contents.
pv	monitor the progress of data through a pipe.
sed	stream editor for filtering and transforming text.
sort	sort lines of text files.
tee	read from standard input and write to standard output and files.
tail	output the last part of file.
tr	translate or delete characters.
uniq	report or omit repeated lines
unzip	list, test and extract compressed files in a ZIP archive
wc	print newline, word, and byte counts for each file.
wget	utility for non-interactive download of files from the Web
zcat	zcat uncompresses either a list of files on the command line or its standard input and writes the uncompressed data on standard output (On some systems, zcat may be installed as gzcat to preserve the original link to compress).
zip	package and compress (archive) files.

The output and the exit code of the executed script will be redirected to the client code. This way, the user would see the contents of the output of the commands like he would see it in a UNIX terminal.

During the execution phase, the server notifies users that are connected to the workflow that it is running, ignoring further requests to run of the same workflow. Imagine if multiple users would request to execute the same graph. The server would execute the same workflow multiple times, wasting available resources. If the workflow is large and complex enough that can take hours to process, it can slow down the processing machines if requested multiple times.

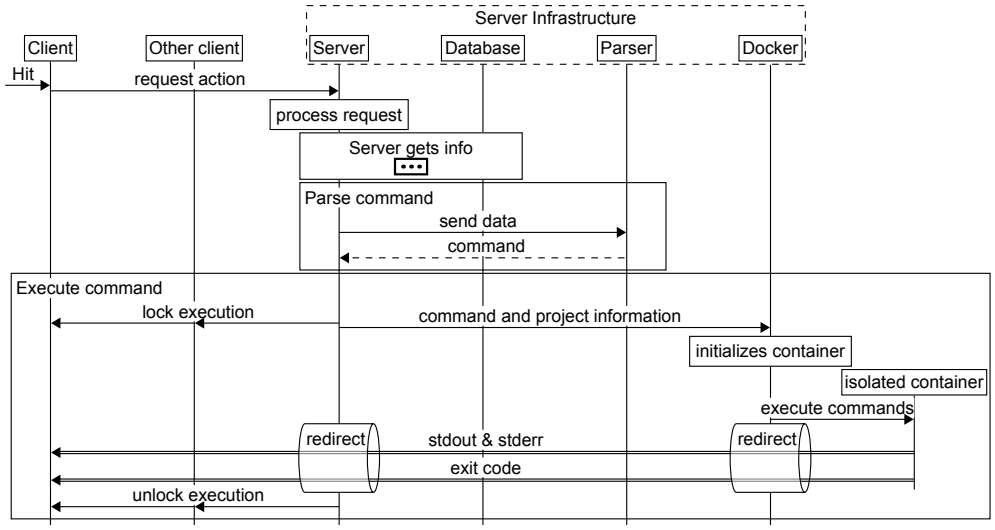

Figure 5.2: Sequence diagram of the execution of a graph

5.3.6 Real-time collaboration

One of the requirements of the application is that users create an modify workflows collaboratively where changes made by a user are immediately noticed by other users working on the same workflow. In short, the prototype should be a real-time collaborative editing tool.

Modern browsers have a standard technology used for those types application called *Web Sockets* [web], a technology that uses a protocol for two-way communication with a remote host, in this case, the server.

Figure 5.3 shows a sequence diagram that shows what happens when a user makes an action in a graph. When an action is made, it notifies the connected peers that he made a specific action (e.g. joining a graph, sending a chat message, etc...) by using the server as a medium.

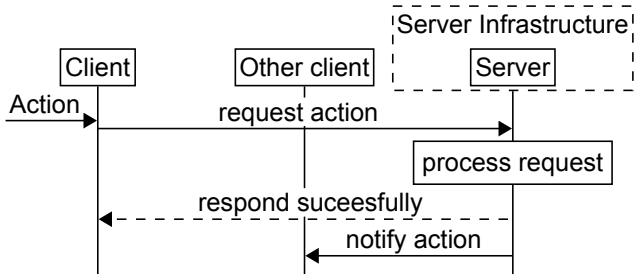

Figure 5.3: Sequence diagram of a simple action

However, many of the actions require modifications of the underlying data (e.g. connecting components, updating components, etc...) which requires the use of more modules in the application. Figure 5.4 shows a sequence of the action done when an action that modifies a graph has been made.

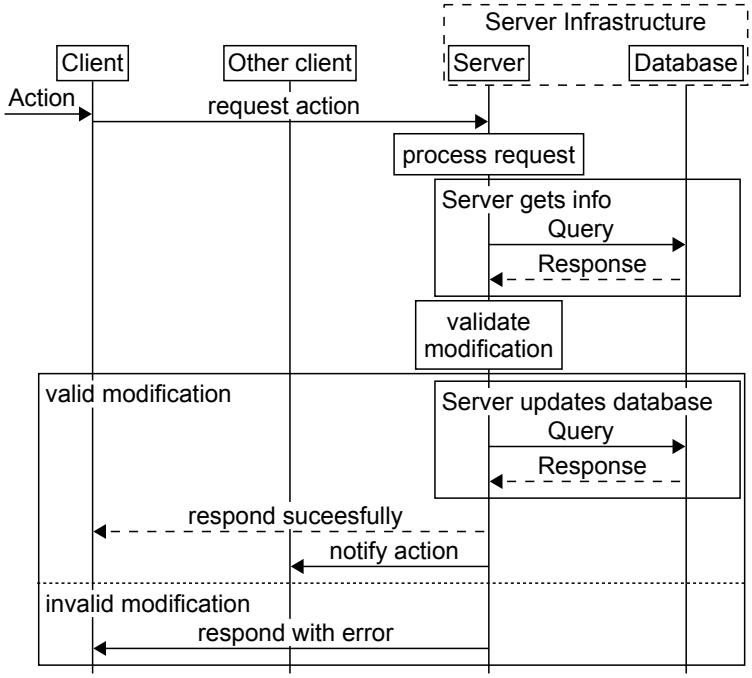

Figure 5.4: Sequence diagram of an action that requires the modification of data.

When the server receives the request, it will first check if the action requires modification of the underlying data. If it does not require modifications, the server just redirects the notification to other users just like the path shown in the figure 5.3. Otherwise, it will query the necessary information to validate the modification. For example, when connecting components, it would validate the connection by checking if it passes it requirements. If the modifications are valid, the server updates the database with the necessary modifications and notifies connected peers about the action made by the client.

5.3.7 Separate filesystems

Each project has a file system were their members have access. This approach is proposed for safety reasons. There is a use-case where multiple projects would need to create files with the same name. Each project will have his own workspace and commands executed in a project would have access to his own workspace.

To exemplify, it is possible that a project that uses a file named *vehicle.txt*, processes it, and saves the result to another file, named *cars.txt* regularly where another project may use a file that uses a file with the same name (*cars.txt*) as a test file to compare the output of a process using the *diff* command.

Figure 5.5 shows a view of the file system, it contains files that are in the project directory. Every time a workflow has been executed, the contents of the filesystem are updated.

The files can be downloaded from or uploaded to the server to be used as input of a workflow, or to download the results of a task. The files can transferred in two ways: by using the download

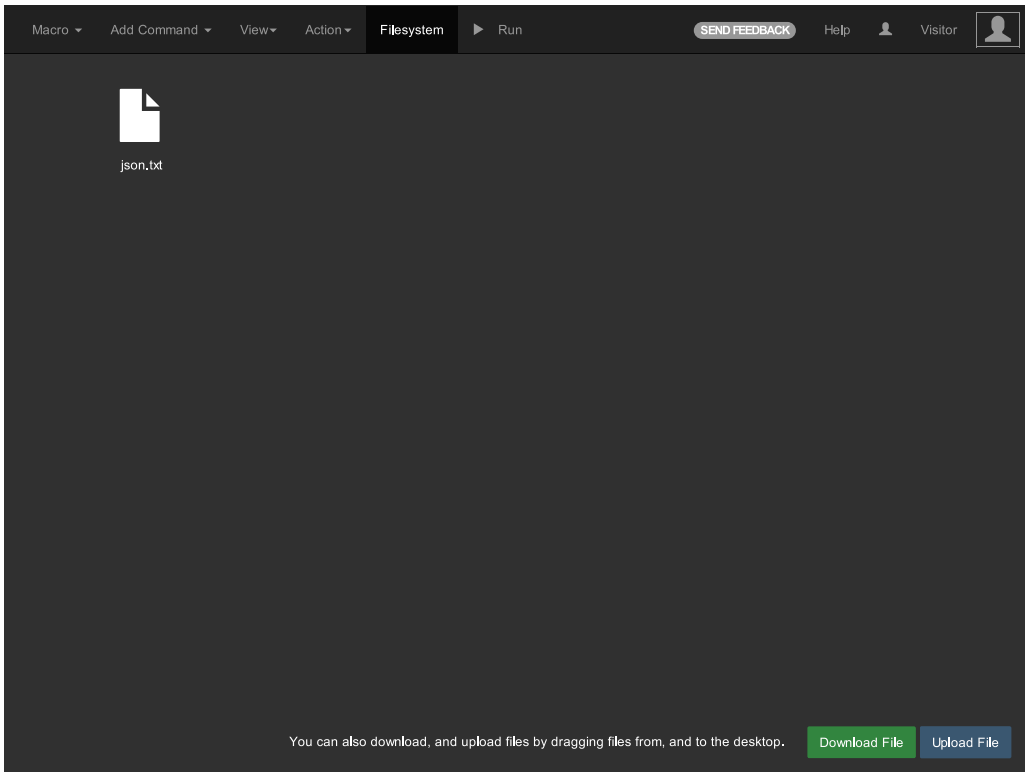

Figure 5.5: View of the filesystem

and upload buttons on the bottom right corner of the view, and by dragging the files from and to the filesystem tab like how a user moves a file from a directory to another with a file manager.

5.3.8 Connected user information and chat system

Although the priority of adding a modal where users could send messages to connected users was low, we believe that identifying the users that are working on the same workflow would allow the user to expect that the graph would be changed without the user interacting with the application if there exist connected users.

There are two ways to differentiate two users: having different usernames or different colors. When a user enters, it is applied a random color of a defined list of defined colors, this way it makes easier to identify 2 users with the same username. The list is sufficiently large so there is a considerably small chance that the applied color would be the same.

Since the implementation of the feature required that a user would be notified immediately when a user enters or leaves into the workflow, a simple chat system was created using a similar logic: notify other users that he made an action. This way, it was fairly easy to add a simple chat system.

5.4 A Tour on the Prototype

We already described most of the concepts defined for the prototype and how they are connected. However, we didn't show how those concepts pass to the user and how they are represented. In this section, we show how we represent the defined concepts.

5.4.1 The main view

Figure 5.6 shows a screen shot of the application where most of the interaction is done. This figure can be spliced in 4 frames:

- **Top bar**. The top bar is located at the top of the screenshot. It contains a menu, buttons to compile and run the workflow, an Help button, an icon with the number of connected users (excluding the user itself), and the user information.
- **Main view (or graph view)**. The main view contains the workflow to be interacted by the user. It is the gray region in the center of the picture
- **User chat system**. The user chat system, located at the right side of the screenshot, contains the information on the connected users, as well as a basic chat system to send quick messages.
- **Console view**. The terminal view, the dark area located at the bottom of the picture displays information about the compiled UNIX Shell code, and displays information about the execution of a workflow.

5.4.2 Creating components

There are two ways to create components. The first way is by clicking on the "Add command" button on the top bar of the application, however these components are restricted to filter components. The other way is trying to connect a component to an empty area. When trying to connect a component, a connection is automatically created where one end will follow the mouse pointer position when the mouse button is pressed. When the mouse button is released on an empty space, a pop-up interface will appear in the position of the mouse. The pop-up contains a list of implemented components, like the "Add command" button, and a text field to write the component that the user wishes to create. The user can create all kind of components using the text field: commands, files and macro components. Rules were created to choose the type of the component to create.

5.4.3 Connecting ports

A connection between an output and an input is displayed using a color for a different type of output connector, their color is similar of the color of the port of a component.

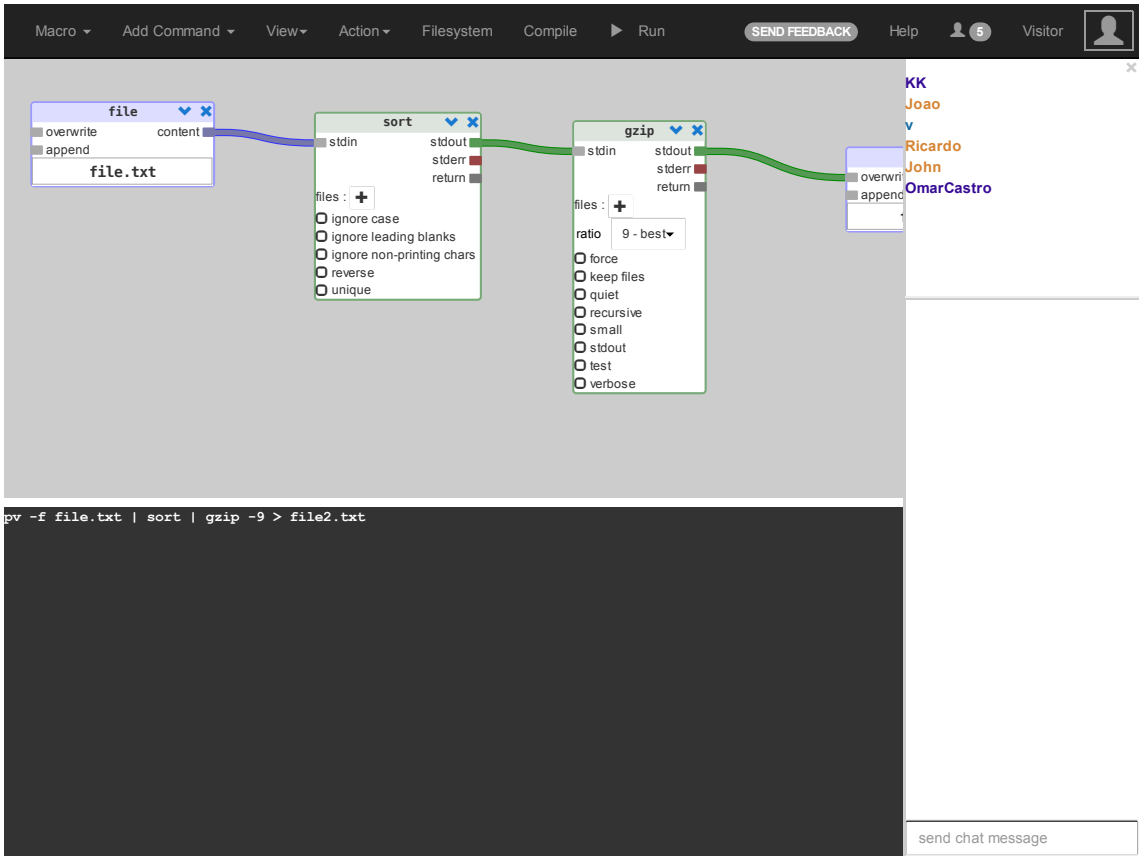

Figure 5.6: A screenshot of the application

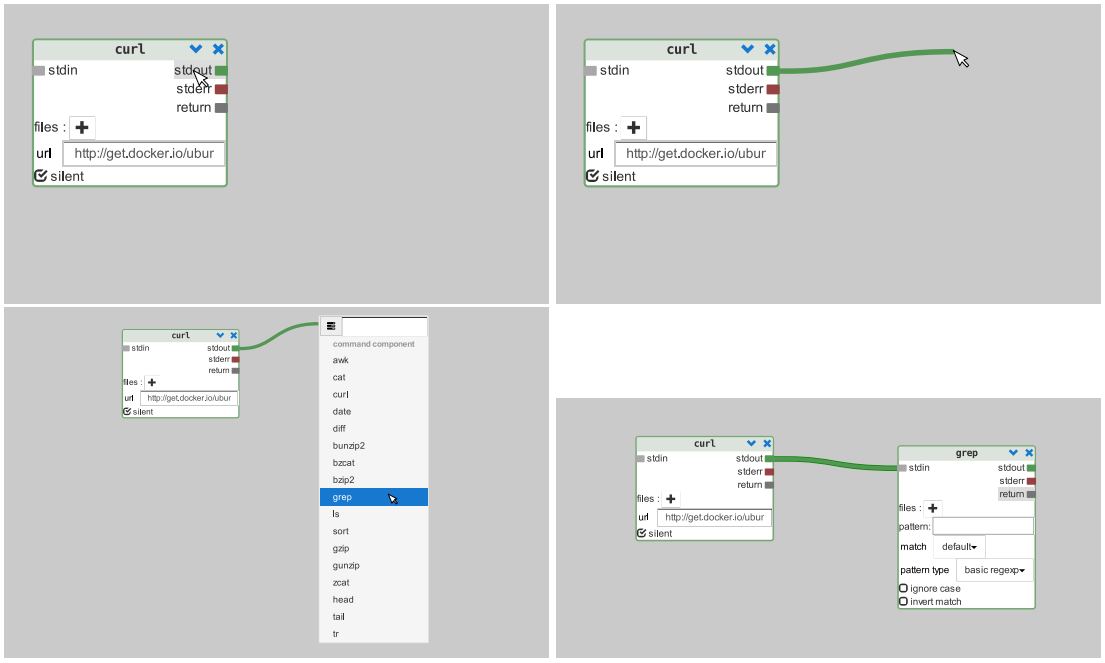

Figure 5.7: An example on how to create a component. By dragging a port to an empty space, a pop-up appears. The order of the image sequence is: top left; top right; bottom left; bottom right.

Initially all connections contained the same colors, but sometimes it was difficult to find a connection when there are many components in a graph when zoomed out. A way to differentiate some connections was developed by using different colors. The colors defined for the connection were based from the output port of a type of a command: *blue* for the output of a file component, *green*, *red* and *dark gray* for the standard output, standard error and the exit code of a filter component respectively, brown for an output of a macro, and *light gray* for other outputs, like the output of an input component.

5.4.4 Implemented components

A set of components were implemented for testing purposes. They were used a fair amount of times to explore and find ways to improve them. Five different types of components were created, each of them with different purposes that will be described within this section. Figure 5.8 shows the visual representation of each type of component.

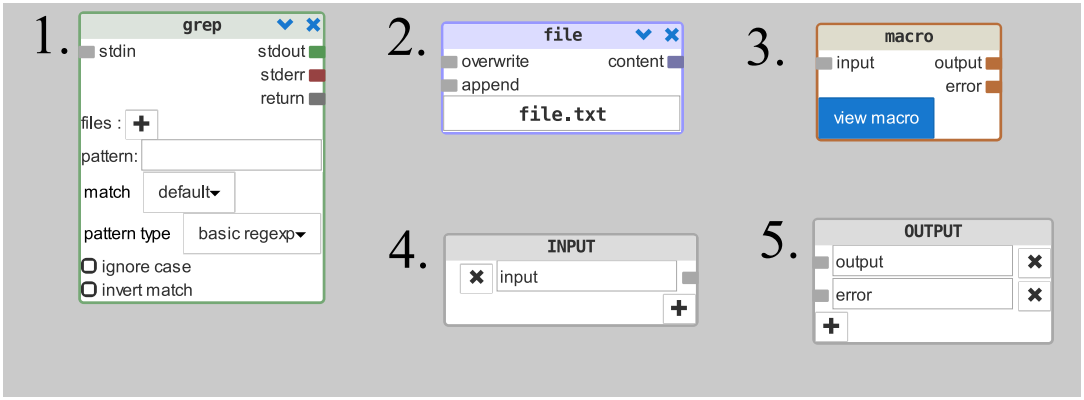

Figure 5.8: Different types of components, the legend is as follows: 1 - filter or command, 2 - file, 3 - macro, 4 - input, 5 - output.

Filter

A filter component, or command component, represents a command in the UNIX Shell. Each kind of command is represented as a filter component with a different interface. The interface can contain 4 types of fields:

- **Flag.** They represent command options that have two states, most commonly the *enabled* or *disabled* state. They are normally located at the bottom of the component, represented as a checkbox.

- **Parameters**. They visually represent text fields. When converting the component to text code, the parameter is not compiled if the parameter is empty. In figure 5.8, the parameter is represented as a text field with a label. In this case, the content of the label is “pattern”.

- **Selectors**. They are represented visually as a select box. They represent options that have two or more states. Some of the options require a parameter. In figure 5.8, the file component contains 2 selectors, the “match” and the “pattern type”.

- **File inputs**. A file input represents an input that a command should read as a file. It is present on filters that represent commands that read multiple files. Each entry contains an input port that can also be connected to an output of another filter component. When compiling a workflow, the inputs connected to a component will represent a named pipe that will be used as a file to be read by the command. In figure 5.8, the plus button next to the “files” label is the representation the file input list. In this case, the command is not supposed to read any files.

  They also have an additional 4 ports, one input port and 3 output ports. The input port represents the standard input of the command. The outputs ports of the components are: output, error, and exit code. The output port represents the *stdout* (standard output) of the command. The error port represents the *stderr* (standard error). And the exit code port represents the return code of the command that is sent after the end of its execution.

File

A file component represents a file in the file system. Each project contains a file system, containing files and data that will be accessed by the isolated container to execute the workflow.

All file components have two inputs and one output. The inputs of a file components are: “overwrite”, which the result of connected components completely overwrites the contents of the file; and “append” where the result is appended to the file. The output of the file component is the content of the file. When a file component is compiled to a UNIX Shell script, the file is read using the `pv` command. The `pv` command sends information about the reading progress of the file through the *stderr* of the command, allowing the user to have an idea about the progress of the script execution, and the time remaining for the content of the file to be completely read.

Macro

A macro component represents a macro in the form of a component. The number of inputs and outputs is variable since the user can add or remove inputs or outputs of a macro. The addition or removal of components can be done when the user is editing a macro. For that reason, a shortcut to view the contents of the macro is included in the component itself.

Input and Output

These components have a set of ports of one type. Output ports on an input component and input for an output. They represent the entry and exit points of a macro. The macro inputs and outputs are included on the output and input components respectively. These components contain interfaces to add and remove input or outputs of a macro.

5.4.5 Creating Macros

As explained before, a macro is a composition of interconnected components. They can be created selecting the "New macro" option on the "Macro" tab at the top bar. When a user clicks it, a modal appears with a form to describe the macro to create.

The modal contains 3 fields:

- **Name**. The name of a macro should contain only alphanumeric characters that is different from the command. This way, it becomes easier to create components using a text field on the component creation pop-up text because it removes ambiguity when creating macros with the same name as the command.
- **Description**. The purpose of the description of the macro is to provide detailed information about the macro, such as its purposes. This is an optional field because the modal can be edited later.
- **Command**. This field allows the user to create the graph automatically by writing a single command line. This field allows the users that are used to write command lines with the keyboard to generate macros quickly. If the field is empty, only the input and output nodes are created, and the workflow would need to be designed manually.

Figure 5.9 shows an example of creating a macro. We start by giving a name, description, and a simple command line.

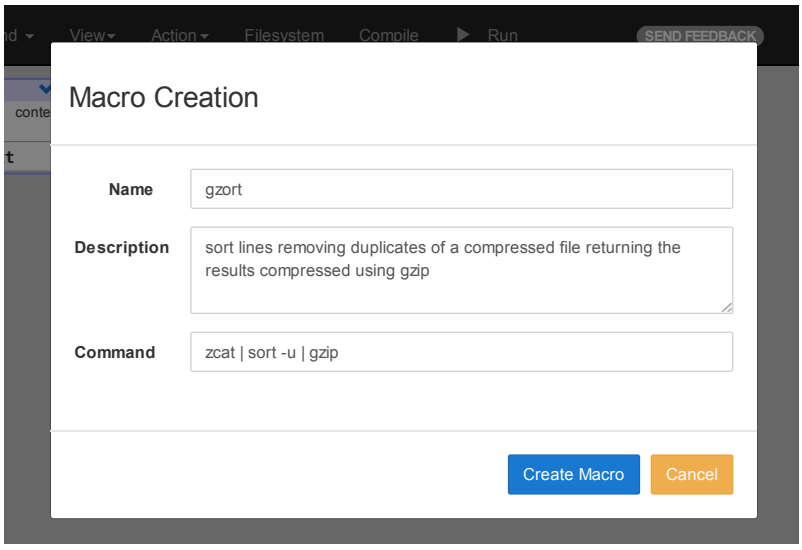

Figure 5.9: Macro creation interface with filled data.

After clicking the "Create Macro" button with the desired command in the "Command" text field, the user would view a new graph with the generated components. The graph contains 2 additional components: the input component, and the output component. They are included in all macros. Figure 5.10 shows the resulting graph of the macro creation.

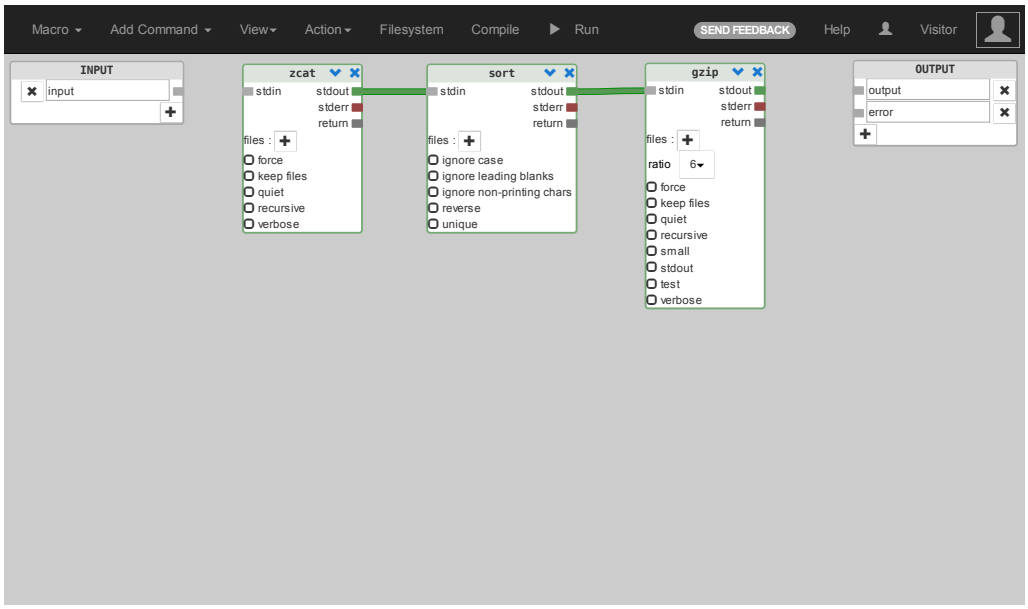

Figure 5.10: Graph view of created macro.

5.4.6 Terminal panel

The terminal panel displays the UNIX Shell script that is the direct translation of the graph. When executing a workflow, the terminal displays the results of the execution: its *stdout*, *stderr* and exit code.

The terminal can be hidden or have its dimensions changed to save space for the user to interact in the main view. To change the height of the panel view, the user would need to drag the thick white line that separates the terminal panel and the main view.

The panel was designed to look similar to a common UNIX Shell terminal since many users that use the UNIX terminal are familiar with the design of a terminal view. However, the panel does have some slight changes, the *stdout* and *stderr* are written with 2 different colors. The *stdout* is printed with a white color and the *stderr* with a red one. The compiled script and the result code of the script are printed with a bolder text to differentiate with the *stdout* and *stderr* of the resulting execution.

5.4.7 Automatic compilation

Having to always compile when changing the model, or knowing what changes were made might give difficulties when debugging a workflow.

Allowing the textual code representation of the graph to be automatically updated can help the user to debug the changes made on the code when changing the graph. This feature can be used as a teaching tool, since a person can learn or remember the purpose of an option.

Figure 5.11 shows a sequence diagram of how the automatic compilation is done. When a change in the graph is made, the client automatically makes a request to the server to get the

resulting command of the workflow. The server would get the identification of the workflow inside the request, query it in the database, compile it, and send the resulting command to the client.

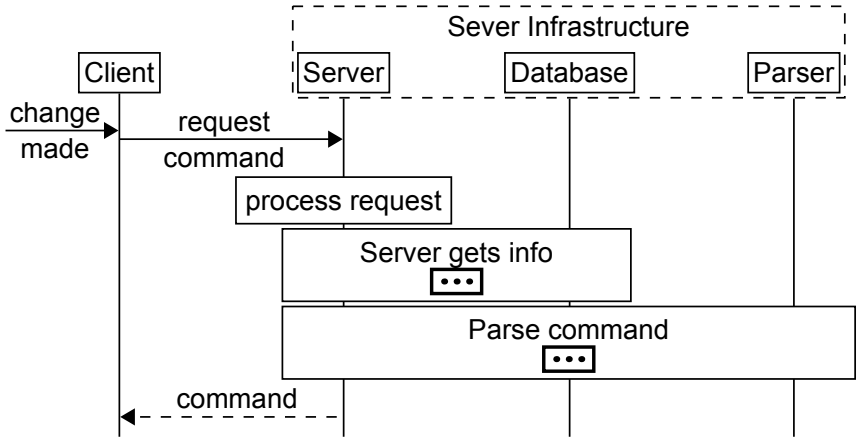

Figure 5.11: Sequence diagram of the execution of the parsing of a workflow.

5.5 Conclusions

In this chapter we explained the architecture of the application, including the implemented features and the reasoning behind it.

A lot of time was invested thinking on how we could represent the information of the components. If we create a script in NoFlo.js that has 40 components, the connections start to overlap. Some blocks get out of the screen and the script becomes difficult and tedious to maintain. The developers of NoFlo created a new concept of compressing the nodes to a single icon [Flo], but they have removed so much information, making it difficult to tell the difference between two components without having to open each component. Our solution was to save space by making the components collapsible hiding any non-relevant information about the component.

There were deviations in the plans, such as adding new concepts (e.g. macro) during the development phase, but we were able to make the application stable enough to be testable by users. We tried to clarify some concepts with a tutorial (Appendix B, page 63). This tutorial can answer to most of the questions that were made by some end-users. A tour of the application was made in order to understand the features of the application.

Solution prototype

Chapter 6

Quasi Experiment

While we were developing our prototype, we tried to collect some feedback from end-users.

We decided to host a demonstration of the application in a server [1], and announce our application at discussion forums about Unix Shell to test the application [2]. The public target to test the prototype were people with experience with the language itself, such as system administrators who use UNIX Shell to process data, and people that are interested in learning how to use UNIX Shell to create data processing tasks.

There were two ways to give feedback: by discussing about the application, or by filling a survey. The latter option allowed us a more easily organization of the feedback.

The result was that most of the feedback was received with discussions. A very small amount of users decided to fill the survey as an option to send feedback, there were even more users that used the built-in chat to send feedback.

That feedback changed some attributes in the application, and others were saved in a backlog. The end-users feedback will be grouped in two lists: a list with items that were solved and other that were not.

6.1 Issues solved

Some issues noticed by end-users were solved, and resulted in following modifications that are part of our prototype:

- **Creating a Graph.** Some of the end-users on the first usage, did not know how to create components nor connect them, because it was not explicit enough on how to use the application. A tutorial (Appendix B, page 63) was created so that users could have basic knowledge on how to use the application.

[1] the link of the demo is paginas.fe.up.pt/~ei08158/thesis/public_demo.php

[2] the application was announced on http://www.reddit.com/r/bash/comments/27e2pg/a_visual_collaborative_unix_shell_interface/ , http://www.reddit.com/r/sysadmin/comments/27gpcd/a_visual_collaborative_unix_shell_interface/ and https://groups.google.com/forum/#!topic/comp.unix.shell/6i3vI9FQcf8

- **Graph compilation generating named pipes.** Even tough the result did not change, end-users thought that adding many named pipes would make the generated code unreadable. The generator was rewritten so that it would add named pipes when necessary.
- **Code execution using arguments.** The code generation was initially made using double quotes. Initially the arguments were sanitized so that it could not execute a unwanted code, however, there were more ways to inject code into the arguments. An end user notified the author about the bug found. After a quick study, the author found that single-quotes weren't interpreted by UNIX Shell interpreters, so a quick patch has been done.
- **Interface layout problems.** The application was developed for any screen resolution, however, it was tested mostly on screens with high resolution. Some end users notified that the application looked weird on low resolution screens, the layout had to be arranged and tested on low resolution screens.

6.2 Issues to be solved

The following issues were noted by end-users but they are still open:

- **Lack of documentation.** Somewhat related to the solved issue of creating graphs, Some of the end-users explained that they had difficulties interacting with the prototype because they were accustomed to use a UNIX terminal for a long time, which turns to be a barrier to use the application. As noted before, a manual has been made, though it only contains basic knowledge on how to interact with the application.
- **Limiting the time of some commands.** One user found out that using the command curl to download large files, made the client side less responsive. While he was interacting with the application, the client received the output of the command, and the application started to become slow, due to the large amount of data that the client received.
- **Creating components through the use of right-click.** Some users found that using the context menu to create components would improve the usability of the application.
- **Not enough number of commands.** This issue was to be expected due to the low priority of adding available commands.

6.3 Survey

The users that opted to use the survey were very small compared to the feedback received in the discussions; 6 responses were made in total. Though it doesn't give any definitive conclusion, we believe that the results are enough to validate our hypothesis.

Most questions given out to users were designed using a Likert Scale [Lik32]. This scaling method consists in a measurement of a positive or negative response to a statement. The questions

in the survey used five-point Likert items, with the following format: 1 - strongly disagree; 2 - somewhat disagree; 3 - neither agree nor disagree; 4 - somewhat agree; 5 - strongly agree.

The questionnaire was divided in three groups:

- **Usability (US)**. The usability questions serve to check if the application was easy to understand.
- **Usability Comparison (CP)**. The Usability Comparison contains questions to compare the usability of the application with the UNIX terminal.
- **Choice of tools (CH)**. This part of the survey contains questions to see which tools the users would choose in different situations. This questionnaire did not have the same format of the five-point Likert items. Instead, it had the same items where the lower values leads to a preference in the UNIX terminal, and the higher values in the developed prototype.

Table 6.1 shows the results of the survey, following the descriptions of each test.

Table 6.1: Survey results

	1 2 3 4 5	$\bar{x}$	σ
US1		4,33	1,03
US2		3,67	1,21
US3		3,83	1,17
US4		3,50	1,22
US5		4,50	1,22
CP1		4,50	1,22
CP2		4,00	1,26
CP3		4,33	1,21
CP4		4,50	0,84
CH1		2,40	1,95
CH2		4,60	0,55
CH3		4,40	0,89
CH4		3,40	1,82

US1. I found the instructions easy to understand.

The experimental group gave a score of $\bar{x} = 4,33, \sigma = 1,03$. The responses gathered were positive and very favourable.

US2. I found the application easy to use.

The users gave a score of $\bar{x} = 3,67, \sigma = 1,21$. The responses gathered were positive and favourable.

US3. I had no trouble creating and executing a graph.

The users gave a score of $\bar{x} = 3,83, \sigma = 1,17$. The responses gathered were positive and favourable.

US4. I had no trouble creating and connecting a macro.

The users gave a score of $\bar{x} = 3,50, \sigma = 1,22$. The responses gathered were positive and favourable.

US5. I had no trouble downloading or uploading files.

The users gave a score of $\bar{x} = 4,50, \sigma = 1,22$. The responses gathered were positive and favourable.

CP1. It is easier to maintain a workflow using the visual graph than a UNIX terminal.

The users gave a score of $\bar{x} = 4,50, \sigma = 1,22$. The responses gathered were positive and favourable.

CP2. It is simpler to create commands with the tool than with a UNIX terminal.

The users gave a score of $\bar{x} = 4,00, \sigma = 1,26$. The responses gathered were positive and favourable.

CP3. It is easier to understand the objective of the tasks with the tool than with a UNIX terminal.

The users gave a score of $\bar{x} = 4,33, \sigma = 1,21$. The responses gathered were positive and favourable.

CP4. It is easier to share a workflow than with a UNIX terminal.

The users gave a score of $\bar{x} = 4,50, \sigma = 0,84$. The responses gathered were positive and favourable.

CH1. When I'm going to work in a group project with experts on UNIX Shell scripting, I would rather use...

The users gave a score of $\bar{x} = 2,40, \sigma = 1,95$, which mean they favoured the UNIX terminal.

CH2. When I'm going to work in a group project with people with no experience on UNIX Shell scripting, I would rather use...

The users gave a score of $\bar{x} = 4,60, \sigma = 0,55$, which mean they favoured greatly the visual application.

CH3. When I'm going to work in a group project with people that I don't know their skills on UNIX Shell scripting, I would rather use...

The users gave a score of $\bar{x} = 4,40, \sigma = 0,89$, which mean they favoured the visual application.

CH4. When teaching someone about processing data using Shell Script, I prefer to use...

The users gave a score of $\bar{x} = 3,40, \sigma = 1,82$, which mean they favoured the visual application.

6.4 Conclusions

This chapter describes the quasi-experiment made to assess the usability of the developed web application. The first phase shows the issues received from the users feedback. Divided in two groups of issues, the one that are solved and the ones that are open. The survey results shows that the application supports the hypothesis that using the application would improve the simplicity and maintainability of the application. The application is far from complete, as it would need to add more tests with end-users, also to solve the issues found by the end-users.

Chapter 7

Conclusions

This dissertation touched several areas that deal with high levels of abstraction, and at the same time it had to be concrete because it deals with end-users. The end-users that used our prototype asked for higher levels of abstraction, so we made a detour that led to some experiences with Unix Shell programming. Despite we didn't use the work we did in that detour, we believe that this was important. In this chapter, we are going to summarize our experience, our contributions, and our thoughts for the future work.

7.1 Overview

In Chapter 1 we introduced our thesis, and presented some important concepts that are related with our thesis. We also explained why we think this thesis is important, an abstract about the problems of the UNIX shell language.

Chapter 2 gives an overview about the background of some important concepts related to the thesis in question. Starting with the general area of this dissertation that is software engineering, to a more specific, such as Visual Dataflow Programming and stream programming.

In Chapter 3, we saw examples of existing applications, that do what we are looking for, such as NoFlo and Blender Composite Nodes, though the purpose of some of the applications is similar, making the application easier to interact by end users; they don't solve the specific use cases of this thesis.

Chapter 4 gives an overview of the problem definition of the thesis, and explains why the current solution doesn't solve the specific use cases of the application.

Chapter 5 shows a documentation of the experimental application to solve the existing problem in the current thesis.

And lastly, chapter 6 describes the quasi-experiments made to validate the hypothesis of the prototype, as well as their results.

7.2 Main Contributions

We believe we have three major contributions: A short-paper, a framework that is independent from Android and a prototype developed for Android that can be used to collect more feedback from end-users.

Prototype

We believe that a typical visual dataflow solution is not enough due to the fact that a great part of the public target is used to write commands using text. We decreased the barrier to those users by initially creating graphs automatically by using paradigms and features used by UNIX terminal.

During the development, we found out that having long workflow would reduce its maintainability, so we tough of a way to reuse tasks: macro. This form of task reuse is achieved through the reuse of the macro as a component.

There was also the idea of sharing macros through multiple projects, however, it was not implemented because the implementation was the task with the lowest priority, so it did stay in the backlog during the development phase.

Our prototype is far from finished, but it is useful to do more tests. It can be used to check what kind of features the end-users need or benefit, if there are other ways to create and connect components, and how easy it is for end-users to create abstract tasks. These are only examples of tests that can be done using our prototype.

Short-paper

Our short-paper was submitted for the 9th international conference on Cooperative Design Visualization and Engineering (CDVE) on 25th April. The paper was accepted on 21st May, and the final version was submitted on 15th June. the short-paper is entitled: Collaborative Web Platform for UNIX-based Big Data Processing.

This short-paper focus on the collaborative part of our framework, where the expert programmers create the parts that can be connected by end-users. At the time we first submitted this short-paper we simply had a work in progress, so that is why we didn't do a full paper.

7.3 Future work

Completing What is Done

We did not complete some of the features that we started. This happened because the priorities have been changing frequently, and some features had to stop to give place to others. We grouped what it needs to be completed in this section.

Updating macro connections

The removal of macro entries and exits doesn't delete and update the connections already made on their respective macro components, there should be rule that was that to remove an input or an output, the related ports of the macro components should not be connected, if that happens, a warning should appear indicating that a connection already exists, this allows to avoid accidentally changes in the graph by removing critical connections, or connections that would change the entire workflow.

Exploring new Solutions

There are features that were never started because we knew that we needed some time We have chosen some of them that we thought that are interesting for future work.

SSH configuration interface. It allows the user to generate keys to execute commands in external computers. This would allow the execution of commands in multiple machines independently of the location, as long as each machine can connect each other.

A special interface to execute external commands in parallel using the parallel command tool, because the parallel tool treats the arguments in a special manner.

A docker image which allows to ease the installation of the application to integrate with the solution.

Improve the connection typing, in other words, add rules that would not allow the connection between two components, for example, trying to modify compressed data without decompressing it first, and trying to decompress the data which was compressed using a different algorithm (e.g. using `bunzip2` on compressed data using `gzip`).

Study the addition of touchscreen support, this allows to make graphs using devices that can be interacted with touch, such as smartphones, tablets or touch-enabled laptops, though there might exist problems related to screen size.

Appendix A

Accepted short-paper

The following paper was submitted for the 11th conference on Cooperative Design Visualization and Engineering (CDVE) on 20th April, 2014. This paper was accepted in 21th May and the final version was submitted at 15th June.

Collaborative Web Platform for UNIX-based Big Data Processing

Omar Castro[1] and Hugo Sereno Ferreira[1,2] Tiago Boldt Sousa[1,2]
{omar.castro, hugo.sereno, tiago.boldt}@fe.up.pt

[1] Department of Informatics Engineering, Faculty of Engineering, University of Porto
[2] INESC Technology and Science (formerly INESC Porto)

Abstract. UNIX-based operative systems were always empowered by scriptable shell interfaces, with a core set of powerful tools to perform manipulation over files and data streams. However those tools can be difficult to manage at the hands of a non-expert programmer.
This paper proposes the creation of a Collaborative Web Platform to easily create workflows using common UNIX command line tools for processing Big Data through a collaborative web GUI.

Keywords: Big Data, UNIX Shell, End-User Programming, Cooperative Programming

1 The Expansion of Data

The volume of data being generated globally has been growing exponentially [1] which is increasing the complexity of managing it using traditional tools. Five dimensions can be identified while evaluating how data is growing [2]: *Volume*, managing terabytes to petabytes and up; *Variety* of different types of data (musics, videos, images, text); *Velocity* of flowing data in all directions, *Variability* inconsistent data flow speeds with periodic peaks; and *Complexity* , correlating and sharing data across entities. Considering the specific needs while dealing with Big Data, new paradigms are being adopted to simplify how information can be processed optimally, both for achieving results faster, as well as to simplify the process of creating new computing pipelines

2 UNIX Shell: A Studied Solution

The UNIX shell adopted concurrency paradigms that are being applied today by modern programming languages [3] since they were introduced [4], e.g. the *stream processing*[5], which is a similar paradigm to the *Pipes and Filters* architecture applied in the UNIX Shell [6].

New tools were created in the UNIX environment that allow the execution of workflows in more conventional ways, like executing commands in a remote

computer using the Secure Shell (SSH) tool, parallel execution of UNIX shell code in one or multiple computer by using the *parallel* command.

However the UNIX shell also has shortcomings, it is difficult to design complex big-data workflows, because not only the user have to study the functionality of each command, which can differ greatly, Furthermore, commands documentation ("man pages") might be complex to read by non expert users.

Due to their native concurrency and distribution and large amount of stream processing commands, UNIX Shells are a relevant platform to consider for Big Data processing.

3 A Block-based Collaborative Solution

The authors propose a collaborative web based framework for creating dataflow-based processing using common UNIX shell commands[3], abstracted as connected block components. The metaphor of a component with inputs and outputs is a common representation in software engineering. Black box testing, per example, uses this metaphor. A user creates components with a set of inputs that transform incoming data, making the result available in an output. These outputs can be connected with inputs of other components, creating a model that is similar to a data-flow diagram. Our approach was implemented as a collaborative web application, with a drag and drop editor to create and connect components. A parsing engine then converts this representation into its textual version, compatible with common UNIX Shells. Figure 1 shows an example of the execution of a workflow and its resulting command line at the bottom.

Being a realtime collaborative framework, the system allows for users to cooperate while building their dataflows. The processing workflow is formed by making typed connections, avoiding possible errors such as infinite loops by creating cycles in the dataflow. The workflow can be created or managed using the dataflow GUI, or using text code that should be compatible with UNIX Shell. All block management, connection, abstraction and information propagation on blocks would be controlled by our framework using a data-flow [7] approach.

3.1 Prototype Implementation

The authors identified a potential in the usage of UNIX tools to execute big data related tasks, while it provides command line interpreters for the users to interact those tools, adding a collaborative aspect of managing it. It was decided to develop a framework for the browser, starting by using technologies that would help developing the application efficiently.

The implementation started in the creation of a parser, to translate text commands to a data representation of a data-flow graph that will be used to generate a graph visually. Since most commands in the UNIX shell code are executables, and many of them are not related to data processing of files, we

[3] the source code is in the link https://github.com/OmarCastro/ShellHive

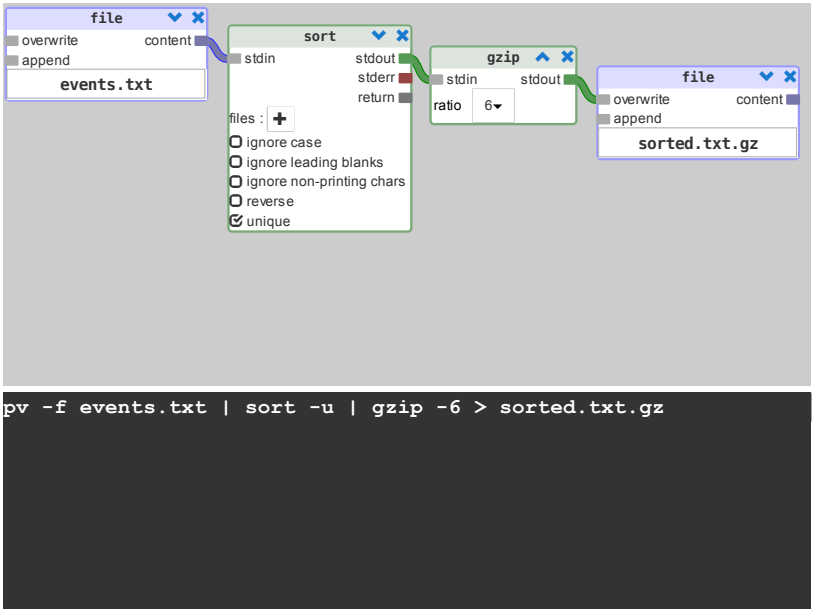

Fig. 1. An example of a command line and it's visual representation

would need to restrict the number of allowed commands. The transformation of text code to visual information has been done for a number of commands the authors believed it was sufficient to advance to the implementation of the GUI, the authors also explored the automatic generation of graphs from a text command, which an expert user can create a workflow by using text commands, since creating workflows with a keyboard can be faster than creating manually with a GUI.

The parse and execution of the workflow in the server has been done initially in a local machine to find security problems. The dataflow would be executed in an isolated container in the server, allowing a safe execution of the workflow. The isolation environment of the commands will be done using the LXC tool [8] to execute the commands without compromising performance and security for those who do the execution. When a user runs a dataflow, every connected peer receives a notification about the execution, and also information about the execution of the application, such as percentage done and time remaining to finish the execution with the help of the *pv*(pipe viewer) command line utility. Every workspace allows uploads and downloads of files, and any generated files after executing the workflow will be added in the workspace so that the user can download them later.

4 Future Work

While the presented prototype can already be used to experiment with the concept, a final product would require further development to increase the products maturity. Another plan is the usage of Docker [9] as the container engine to execute the commands in a sandboxed container. The engine uses the LXC tool to create said containers. Then we are going to begin to implement the user management in the server and as well as the technologies required to make the application a collaborative realtime web application. We also plan to include an interface to manage external servers, which allow the execution of commands in multiple machines, or in a remote server.

After all phases are completed, we plan to test both our end-user as well as expert programmers frameworks. Both expert programmers and end-users will test the application. There are two ways to do this test, one way is to host the application in a server and ask interested people in the web community to use the application and ask for feedback, Other way is to collect data with interviews, locally or hosted in a server, we will ask for expert programmers to develop a workflow and see the difficulties they will find. With this test, we intend to get answers for: Do we provide sufficient debug options? Is our application intuitive? Then we will ask them to compare their experiences with the usage only text commands. After those tests, we will analyze the data, fix some bugs and test again. Final goal will be to open-source our work and freely distribute it within the community, allowing private deployments of the system.

References

1. Big data and the creative destruction of todays business models, strategic it article, a.t. Kearney.
2. Big Data Meets Big Data Analytics, SAS White Paper, 2011
3. Kay A. Robbins and Steven Robbins. 1995. Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
4. Stephen R Bourne. An introduction to the UNIX shell. Bell Laboratories, 1978.
5. T.W. Bartenstein and Y.D. Liu. Green streams for data-intensive software. In Software Engineering (ICSE), 2013 35th International Conference on, pages 532541. IEEE Press, 2013.
6. T. Scheibler, F. Leymann, and D. Roller. Executing pipes-and-filters with workflows. In Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on, pages 143148, May 2010.
7. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. 36 (March 2004) 134
8. LXC - Linux Containers
   https://linuxcontainers.org [Online; acessed 18-April-2014].
9. Homepage - Docker: the Linux container engine
   https://www.docker.io [Online; acessed 19-April-2014].

Appendix B

Tutorial

The following image are screenshots captured directly from the Firefox web browser, when starting the demo of the prototype.

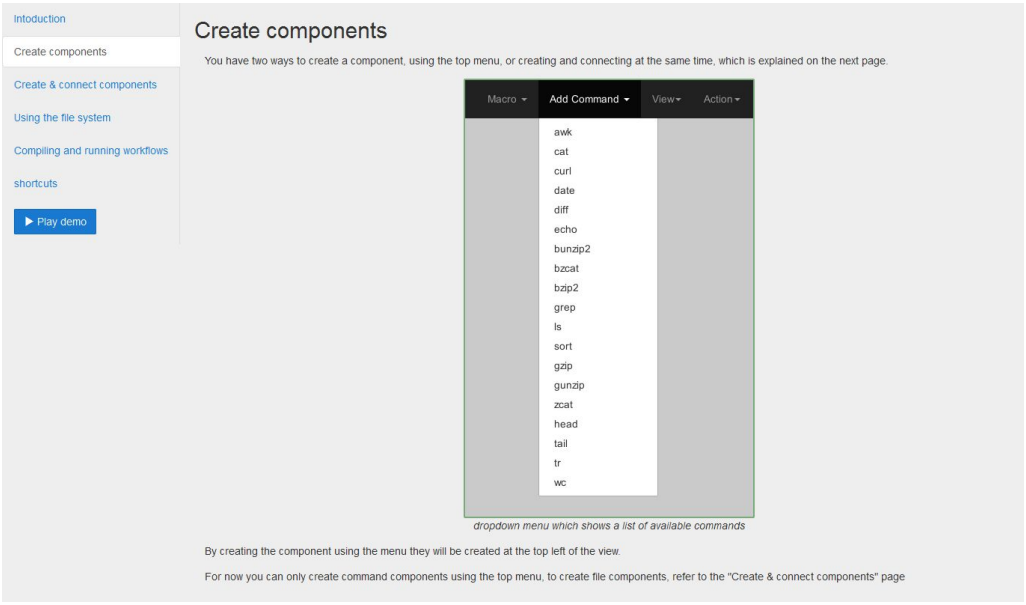

Figure B.1: Screenshot of the *Create components* tutorial

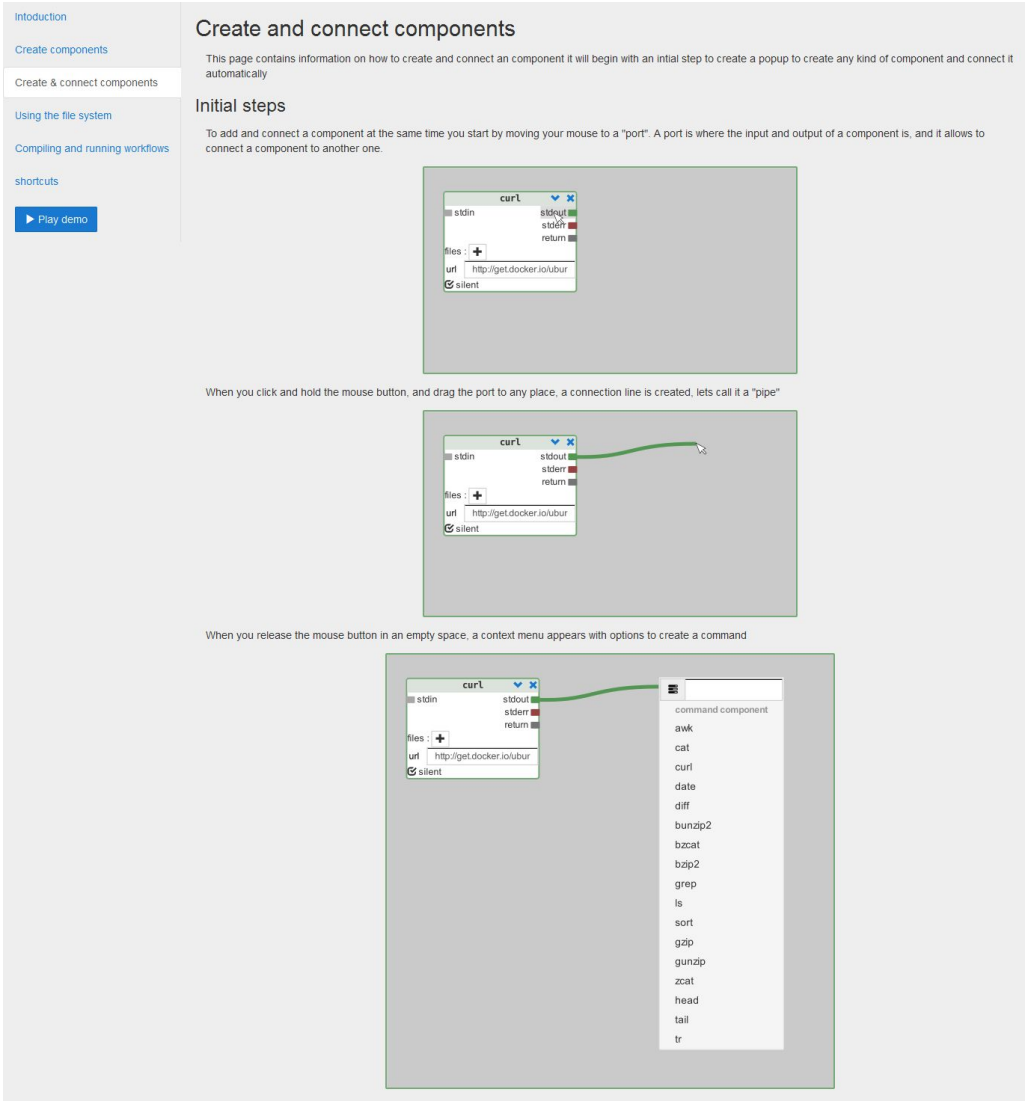

Figure B.2: Screenshot of the *Create and connect components* tutorial, top part

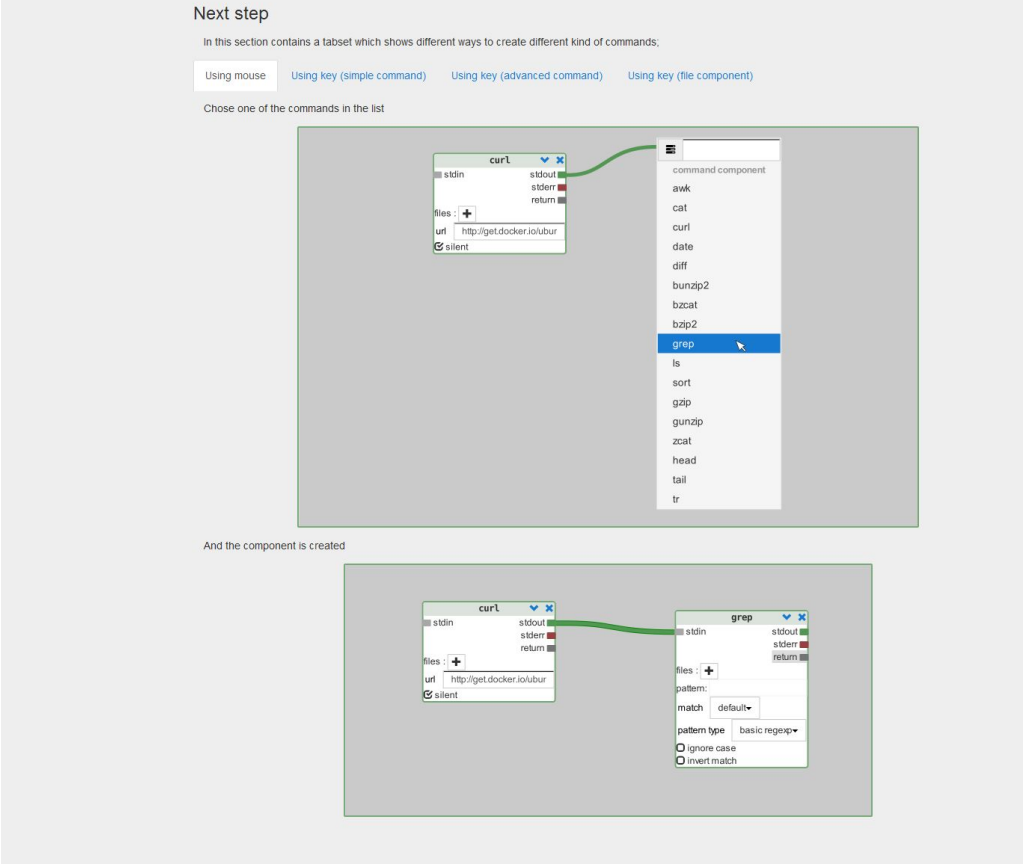

Figure B.3: Screenshot of the *Create and connect components* tutorial, bottom part

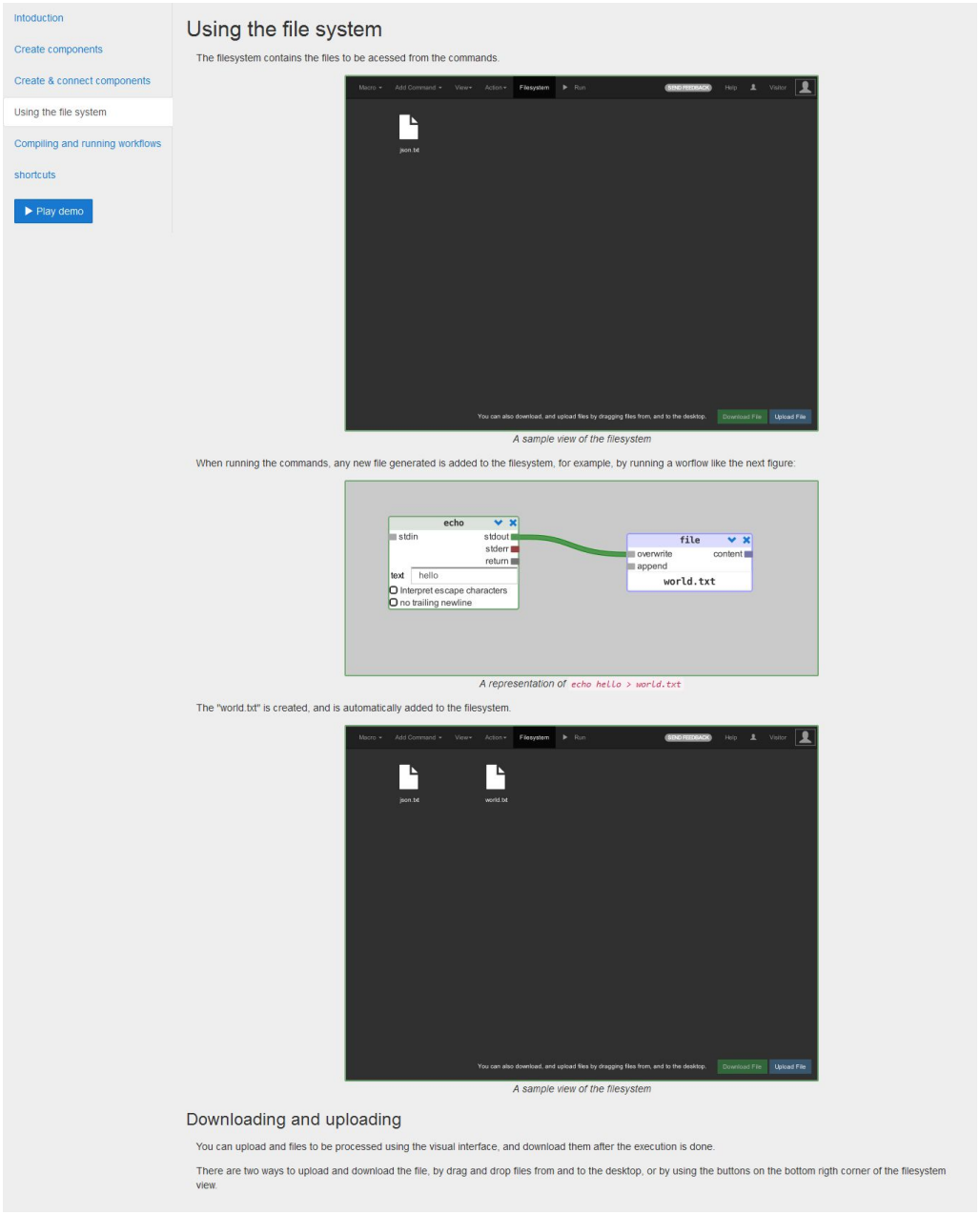

Figure B.4: Screenshot of the *Using the file system* tutorial

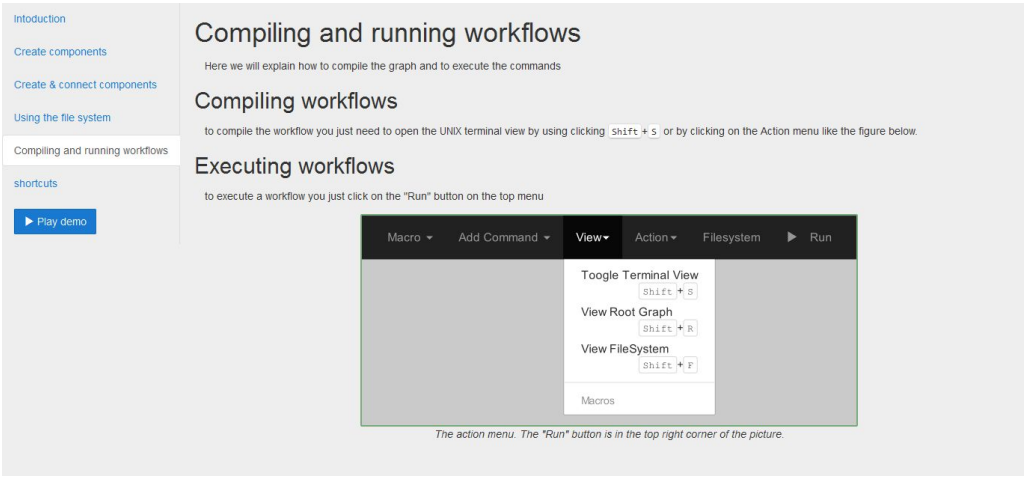

Figure B.5: Screenshot of the *Compiling and runnig workflows* tutorial

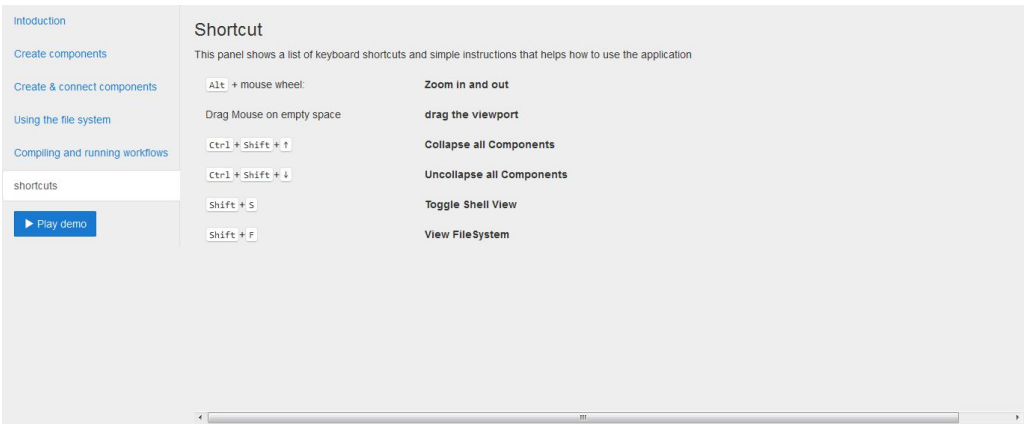

Figure B.6: Screenshot of the *Shortcuts* tutorial

References

[AA92] K. S. R. Anjaneyulu and John R. Anderson. The advantages of data flow diagrams for beginning programming. In *Proceedings of the Second International Conference on Intelligent Tutoring Systems*, ITS '92, pages 585–592, London, UK, UK, 1992. Springer-Verlag.

[Anga] Angularjs — superheroic javascript mvw framework. http://angularjs.org/. [Online; accessed 23-June-2014].

[Angb] Angularui for angularjs. http://angular-ui.github.io/. [Online; accessed 26-June-2014].

[Aß03] Uwe Aßmann. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33 – 41, 2003. {SC} 2003, Workshop on Software Composition (Satellite Event for {ETAPS} 2003).

[bas] Bash reference manual: Major differences from the bourne shell. http://www.gnu.org/software/bash/manual/html_node/Major-Differences-From-The-Bourne-Shell.html. [Online; accessed 2-February-2014].

[BB94] Margaret M Burnett and Marla J Baker. A classification system for visual programming languages. *Journal of Visual Languages & Computing*, 5(3):287–300, 1994.

[BD97] Marat Boshernitsan and Michael Downes. Visual programming languages: A survey, 1997.

[BD04] Marat Boshernitsan and Michael Sean Downes. *Visual programming languages: A survey*. Citeseer, 2004.

[BL13] T.W. Bartenstein and Y.D. Liu. Green streams for data-intensive software. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 532–541. IEEE Press, 2013.

[Ble14] Blender node compositor wiki. http://wiki.blender.org/index.php/Doc:2.6/Manual/Composite_Nodes, 2014. [Online; accessed 26-January-2014].

[Bor90] Kjell Borg. Ishell: A visual unix shell. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 201–207, New York, NY, USA, 1990. ACM.

[Bou78] Stephen R Bourne. *An introduction to the UNIX shell*. Bell Laboratories, 1978.

[Bur01] Margaret M. Burnett. *Visual Programming*. John Wiley & Sons, Inc., 2001.

[CG11] Philip T. Cox and Simon Gauvin. Controlled dataflow visual programming languages. In *Proceedings of the 2011 Visual Information Communication - International Symposium*, VINCI '11, pages 9:1–9:10, New York, NY, USA, 2011. ACM.

[data] Big data and the creative destruction of today's business models - strategic it article - a.t. kearney. http://www.atkearney.com/strategic-it/ideas-insights/article/-/asset_publisher/LCcgOeS4t85g/content/big-data-and-the-creative-destruction-of-today-s-business-models/10192. [Online; accessed 10-February-2014].

[datb] Big data definition - mike2.0, the open source methodology for information development. http://mike2.openmethodology.org/wiki/Big_Data_Definition. [Online; accessed 10-February-2014].

[DK82] Alan L Davis and Robert M Keller. Data flow program graphs. 1982.

[EAG+07] Mattan Erez, Jung Ho Ahn, Jayanth Gummaraju, Mendel Rosenblum, and William J Dally. Executing irregular scientific applications on stream architectures. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 93–104. ACM, 2007.

[Flo] Flowhub v0.1.8. http://app.flowhub.io/#example/7804187. [Online; accessed 23-June-2014].

[GCTR08] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: Programming general-purpose multicore processors using streams. *SIGARCH Comput. Archit. News*, 36(1):297–307, 2008.

[GR04] Jayanth Gummaraju and Mendel Rosenblum. Stream processing in general-purpose processors. In *Proceeding of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[Hie] Hierarchical layout style. http://docs.yworks.com/yfiles/doc/developers-guide/incremental_hierarchical_layouter.html. [Online; accessed 23-June-2014].

[HLR08] Thomas Hettel, Michael Lawley, and Kerry Raymond. Model synchronisation: Definitions for round-trip engineering. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2008.

[hol] Alan kay demos grail - youtube. http://www.youtube.com/watch?v=QQhVQ1UG6aM. [Online; accessed 28-January-2014].

[IBM14] Ibm data lifecycle of big data environments. http://www-01.ibm.com/software/data/optim/data-lifecycle-big-data/, 2014. [Online; accessed 27-January-2014].

[Jis] Jison. http://zaach.github.io/jison/. [Online; accessed 23-June-2014].

[Lap07] Philip A Laplante. *What every engineer should know about software engineering*. CRC Press, 2007.

[Lay14] Chapter 5. automatic graph layout. http://docs.yworks.com/yfiles/doc/developers-guide/layout.html, 2014. [Online; accessed 23-June-2014].

[Lik32] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[LO87] Clayton Lewis and Garay Olson. Can principles of cognition lower the barriers to programming? In *Empirical Studies of Programmers:Ssecond Workshop*, volume 17, pages 248–263. Ablex Publishing Corp., 1987.

[Mye86] B. A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. *SIGCHI Bull.*, 17(4):59–66, April 1986.

[Mye90a] Brad A Myers. Taxonomies of visual programming and program visualization. In *Journal of Visual Languages & Computing*, volume 1, pages 97–123. Elsevier, 1990.

[Mye90b] Brad A Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.

[nofa] clock demo - dataflow-noflo. http://noflojs.org/dataflow-noflo/demo/clock.html. [Online; accessed 10-February-2014].

[nofb] Noflo | flow-based programming for javascript. http://noflojs.org/. [Online; accessed 15-February-2014].

[NoF14] Noflo kickstarter: The hacker's perspective | javalobby. http://java.dzone.com/articles/noflo-kickstarter-hackers, 2014. [Online; accessed 26-January-2014].

[pipa] Pipes - about the editor. http://pipes.yahoo.com/pipes/docs?doc=editor. [Online; accessed 23-June-2014].

[pipb] Pipes - documentation. http://pipes.yahoo.com/pipes/docs. [Online; accessed 23-June-2014].

[RDK+98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 3–13, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[SLR10] T. Scheibler, F. Leymann, and D. Roller. Executing pipes-and-filters with workflows. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 143–148, May 2010.

[sof] Computing degrees & careers » software engineering. http://computingcareers.acm.org/?page_id=12. [Online; accessed 23-June-2014].

[SS12] Sachchidanand Singh and Nirmala Singh. Big Data analytics. In *2012 International Conference on Communication, Information & Computing Technology (ICCICT)*, pages 1–4. IEEE, October 2012.

[Sys] The Unix System. The unix system – history and timeline – unix history. http://www.unix.org/what_is_unix/history_timeline.html. [Online; accessed 1-February-2014].

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R.Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer Berlin Heidelberg, 2002.

[web] Websocket.org | about websocket. http://www.websocket.org/aboutwebsocket.html. [Online; accessed 23-June-2014].

[WH00] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5):242–252, May 2000.

[YL06] Tatu Ylonen and Chris Lonvick. The secure shell (ssh) protocol architecture. 2006.